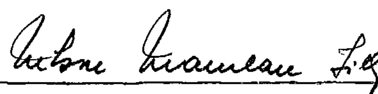
"ARMAZENAMENTO, RECUPERAÇÃO E TRATAMENTO DE MATRIZES ESPARSAS DE GRANDE PORTE"

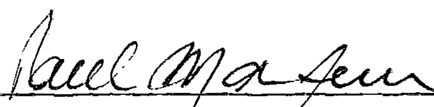
Adelmiro Diniz Costa

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS À OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.)

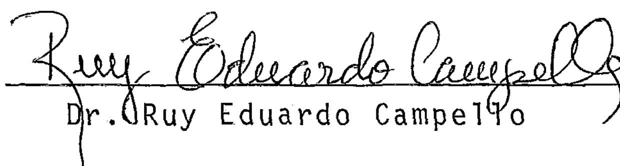
Aprovada por:



Prof. Nelson Maculan Filho
(Presidente da Banca)



Prof. Paulo Mattos de Lemos



Dr. Ruy Eduardo Campello

Rio de Janeiro, RJ - BRASIL

Dezembro de 1980

COSTA, Adelmiro Diniz

Armazenamento, Recuperação e Tratamento de Matrizes Esparsas.

Rio de Janeiro, COPPE-UFRJ, 1980, X, 164 f.

Tese: Mestre em Ciências (Engenharia de Sistemas e Computação).

1. Matrizes Esparsas   2. Armazenamento de Matrizes

3. Tratamento de Matrizes   4. Operações com Matrizes

I. Universidade Federal do Rio de Janeiro - COPPE

II. Título (série)

AGRADECIMENTOS

. Ao Professor Nelson Maculan Filho, que me deu um apoio incondicional para que eu pudesse desenvolver este trabalho, e a quem eu considero, acima de tudo, um grande amigo.

. Ao Professor Paulo Mattos de Lemos, pelo incentivo e ajuda em muitas etapas de minha vida na COPPE-UFRJ.

. Ao Doutor Ruy Eduardo Campello, de FURNAS, por sua revisão criteriosa e objetiva de todo o texto e anexos.

. À minha esposa Mary Alicia Bruckner Costa, que soube criar condições para que pudéssemos concluir mais esta etapa de nossa vida, apesar de todos os problemas que tivemos que enfrentar.

SINOPSE

O objetivo deste trabalho de pesquisa é o estudo de algorítmos apropriados para matrizes esparsas de grande porte.

A pergunta de pesquisa do trabalho é a seguinte: pode-se projetar algorítmos apropriados para matrizes esparsas de grande porte, utilizando linguagens de alto nível, que permitam uma redução do uso de memória principal e de operações de paginação?

A hipótese que se deseja provar ser verdadeira é a seguinte: é possível desenvolver e implementar algorítmos apropriados para matrizes esparsas de grande porte, utilizando linguagens de alto nível como o FORTRAN, que permitam uma redução do uso de memória principal e de operações de paginação quando comparados com os algorítmos tradicionais para tratamento de matrizes.

O presente trabalho está estruturado em seis capítulos, que apresentam o assunto central de forma didática para leitores de diversas áreas da ciência, que se utilizem de matrizes esparsas em suas atividades. O primeiro capítulo, chamado de Introdução, apresenta um histórico do problema e discute sua importância prática para a comunidade. O segundo capítulo, chamado Conceitos Básicos, discute a filosofia de memória virtual, a estrutura de vetores e matrizes sob o ponto de vista conceitual e finalmente como as matrizes são armazenadas em FORTRAN. O terceiro capítulo, chamado de Álgebra

de Matrizes, estuda as principais operações com matrizes tendo por objetivo reduzir as operações de paginação realizadas pelo sistema operacional de um computador. O quarto capítulo, chamado Sistemas de Equações Lineares, estuda o que são os sistemas de equações lineares, os métodos de solução de tais sistemas que são chamados de métodos de eliminação ou diretos e os chamados iterativos, terminando com uma comparação entre eles. O quinto capítulo, chamado de Matrizes Esparsas, discute a filosofia de matrizes esparsas, apresenta algumas alternativas de armazenamento e recuperação, seleciona uma das alternativas e discute as principais operações com matrizes esparsas em problemas práticos usando os conceitos apresentados nos capítulos anteriores, mostra as opções de otimização do compilador FORTRAN que podem ser utilizadas por qualquer usuário e finalmente enumera uma relação dos algorítmos que foram implementados, com base em todo o arcabouço teórico - empírico apresentado no corpo do trabalho, com o objetivo de provar a veracidade da hipótese inicial da pesquisa. Finalmente, temos o capítulo sexto, chamado de Conclusões e recomendações para implantação de códigos em Programação Linear, onde são apresentadas todas as conclusões dos autores. Além das conclusões explicitamente mencionadas no texto, o presente tabalho de pesquisa apresenta um resultado prático, que é a disponibilidade de um sistema composto por um conjunto de subrotinas escritas em FORTRAN, que estão implementadas no sistema B6700 do Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro e que podem ser facilmente convertidas para outras instalações. Maiores informações podem ser obtidas diretamente com os autores na COPPE/UFRJ.

ABSTRACT

The objective of this research is to study large sparse matrices suitable algorithms.

The research question of this work is the following: may one create large sparse matrices suitable algorithms, using high level programming languages, that reduce the ammount of main memory usage and paging activity?

The hypothesis that we want to prove to be true is this: it is possible to develop and implement large sparse matrices suitable algorithms, using high level programming languages like FORTRAN, that reduce the ammount of main memory usage and paging activity when compared with the traditional handling matrices algorithms.

This work is structured in six chapters, wich presents the main subject in a didactic way to the readers of several fields in science, that use sparse matrices. The first chapter, named Introduction, presents the history of the problem and discusses its practical importance to the community. The second chapter, called Basic Concepts, discusses the philosophy of virtual memory, the vector and array structure under the conceptual point of view and finally how the array are stored in FORTRAN. The third chapter, called Matrix Algebra, shows the main matrix operations with the objective of reducing the paging activity executed by the operating system. The fourth chapter, called Linear Equations Systems, discusses what are

the linear equations systems, the methods used to solve such systems which are called elimination or direct methods and the iterative methods, and finishing with a comparison between those types of methods. The fifth chapter, called Sparse Matrices, discusses the philosophy of Sparse Matrices, presents some alternatives of storing and retrieval, selects one of the alternatives and discusses the main operations with sparse matrices in practical problems using the concepts shown in the previous chapters, presents the optimization options of the FORTRAN compiler wich may be used by any user and finally shows a list of all the algorithms implemented, based upon the concepts discussed in this research, with the objective to prove the truth of the research hypothesis. Finally we have the sixth chapter, called Conclusions and Recomendations for Linear Programming codes implementation, where all the authors conclusions are presented. Besides the conclusions explicitly written in the text, this research has one practical result, a system of subroutines written in FORTRAN, implemented at the Burroughs 6700 computer system, at Nucleo de Computação Eletrônica in Rio de Janeiro Federal University. That package can be converted easily to other computer systems. More detailed informations can be obtained with the authors at COPPE/UFRJ.

ÍNDICE

1. INTRODUÇÃO 1

2. CONCEITOS BÁSICOS 5
   - 2.1 Memória Virtual 5
   - 2.2 Memória Virtual no B6700 11
   - 2.3 Vetores e Matrizes 21
   - 2.4 Armazenamento de Matrizes em FORTRAN 25

3. ÁLGEBRA DE MATRIZES 34
   - 3.1 Definições 35
   - 3.2 Adição de Matrizes 38
   - 3.3 Multiplicação Escalar 39
   - 3.4 Produto de Matrizes 40
   - 3.5 Tipos Especiais de Matrizes 45
   - 3.6 Tratamento de Matrizes por Partição em blocos 48
     - 3.6.1 Multiplicação de Matrizes Sub-divididas em Blocos 54

4. SISTEMAS DE EQUAÇÕES LINEARES 58
   - 4.1 Noções Preliminares 59
   - 4.2 Métodos de Eliminação 61
     - 4.2.1 Método de "GAUSS" 64
     - 4.2.2 Método de Eliminação de GAUSS-JORDAN 69
     - 4.2.3 Problemas de Arredondamento 71
   - 4.3 Métodos Iterativos 78
     - 4.3.1 Método de "JACOBI" 81
     - 4.3.2 Método de GAUSS-SEIDEL 84
     - 4.3.3 Problemas de Convergência dos Métodos 86
   - 4.4 Considerações Sobre os Métodos 88

5. MATRIZES ESPARSAS 90
   - 5.1 O Que São Matrizes Esparsas 91
   - 5.2 Alternativas de Armazenamento 96
     - 5.2.1 Uso de Listas Linkadas 98
       - 5.2.1.1 Aspectos Computacionais 102

5.2.2 Outros Métodos de Compressão 106
5.2.2.1 Armazenamento Comprimido em Vetor .. 107
5.2.2.2 Armazenamento Com Três Vetores 108
5.2.2.3 Armazenamento Com Dois Vetores 110
5.2.2.4 Armazenamento Com Um Vetor Compactado 112
5.3 Operações com matrizes esparsas 115
5.3.1 Leitura de Uma Matriz 117
5.3.2 Impressão de Uma Matriz Esparsa 122
5.3.3 Impressão do Vetor de Controle 122
5.3.4 Gravação da Matriz em Memória Auxiliar 123
5.3.5 Leitura da Matriz da Memória Auxiliar 123
5.3.6 Cálculo da Transposta da Matriz 123
5.3.7 Mover Uma Matriz Esparsa Para Outra 124
5.3.8 Copiar Uma Linha da Matriz 124
5.3.9 Copiar Uma Coluna da Matriz 125
5.3.10 Trocar o Conteúdo de Duas Linhas 125
5.3.11 Trocar o Conteúdo de Duas Colunas 125
5.3.12 Permutar as Linhas de Uma Matriz 126
5.3.13 Permutar as Colunas de Uma Matriz 126
5.3.14 Soma de Duas Matrizes Esparsas 126
5.3.15 Somar Uma Linha I1 a Uma Linha I2 n Vezes..... 127
5.3.16 Somar Uma Coluna C1 a Uma Coluna C2 n Vezes .. 127
5.3.17 Multiplicar Uma Linha da Matriz por Um Escalar 128
5.3.18 Multiplicar Uma Coluna da Matriz por Um Escalar 128
5.3.19 Multiplicar Todos os Elementos de Uma Matriz Por Um Escalar 128
5.3.20 Multiplicação de Duas Matrizes Esparsas 129
5.3.21 Armazenamento de Informações no Vetor de Controle 129
5.3.22 Recuperação de Informações do Vetor de Controle 130
5.3.23 Inversão de Uma Permutação 130
5.3.24 Inversão de Uma Matriz Esparsa 130
5.4 Opções de Otimização no FORTRAN 132
5.4.1 Opção de Otimização do Compilador 132
5.4.2 Segmentação de Matrizes 133
5.4.3 Segmentação do Código Gerado 134

5.4.4 Segmentação Controlada pelo Usuário 139
5.4.5 Compilações Separadas 140
5.4.6 Uso do 'Vector Mode' 141
5.5 Algorítmos Implementados 143

6. CONCLUSÕES E RECOMENDAÇÕES PARA IMPLANTAÇÃO DE CÓDIGOS EM PROGRAMAÇÃO LINEAR 149

BIBLIOGRAFIA 160

1. INTRODUÇÃO

A solução de muitos problemas lineares em ciência e engenharia se reduz ao problema matemático de resolver um conjunto de equações lineares. Existem dois tipos principais de métodos para solucionar sistemas lineares: o método direto e o iterativo. O método direto busca a solução após um número finito de diferentes operações. O método iterativo começa com uma aproximação da solução e por meio do uso repetido do mesmo conjunto de operações ajusta o valor original da aproximação ao valor desejado da solução. Métodos iterativos tiveram um grande desenvolvimento com a utilização de computadores, que podem realizar milhares de operações num reduzido período de tempo.

Os métodos iterativos usados originalmente para solucionar sistemas lineares são aqueles do século XIX empregados pelo matemático alemão Karl Gustav Jacob Jacobi e pelo astrônomo alemão P.L. Seidel (Gauss-Seidel). Estes métodos conduzem à solução de um sistema linear que pode ser representado em sua forma matricial por $Ax = b$, onde A é uma matriz quadrada de ordem n com elementos reais, e x e b são vetores com n elementos reais. A matriz A, por hipótese, é não-singular; isto é: $|A| \neq 0$, onde $|A|$ é o determinante de A. O problema é encontrar os elementos de vetor x, dados os elementos de A e b. Este problema, aparentemente simples, pode ser de difícil solução na prática, quando tratamos com matrizes de dimensões maiores que aquelas apresentadas como exemplos nos livros texto. Podemos empregar computadores para tratar estes

casos mais complexos, poderiam argumentar alguns. Não devemos esquecer, no entanto, que os computadores são máquinas finitas e que sua memória principal nem sempre pode armazenar as matrizes que alguns usuários gostariam de manipular, de uma só vez. Nestes casos a utilização de memória auxiliar é praticamente imperativa, conduzindo a algorítmos complexos e poucos eficientes para a solução de sistemas lineares. Recentemente foram desenvolvidos sistemas capazes de gerenciar recursos de memória principal e auxiliar, facilitando o desenvolvimento de grandes aplicações. Os benefícios trazidos por tal desenvolvimento podem ser avaliados em termos de uma produtividade maior em programação, legibilidade, facilidade de manutenção e clareza de documentação. Esta característica dos sistemas é chamada de "memória virtual", onde um sistema interage, sem intervenção do usuário, entre a memória principal e dispositivos de memória auxiliar simulando uma memória principal de grande capacidade de armazenamento. Mesmo que estejamos usando sistemas deste tipo, devemos ter determinados cuidados para não perdermos a eficiência dos algorítmos usados nos programas de aplicação. Isto pode ocorrer quando o número de interações entre memória principal e auxiliar aumenta acima de um determinado nível considerado aceitável e boa parte do tempo é consumido em operações de entrada e saída, condição esta que é conhecida como "thrashing". Este problema é particularmente importante no caso de programas que manipulam grandes matrizes, como no caso de solucionar sistemas de equações lineares para as mais diversas áreas de conhecimento humano.

Este trabalho trata o caso de armazenamento e

operações com matrizes de grande porte do tipo "esparsa". Foi utilizado o computador BURROUGHS B6700 do Núcleo de Computação Eletrônica (NCE) da Universidade Federal do Rio de Janeiro para testar os algorítmos apresentados no decorrer deste trabalho. O B6700 é um equipamento que trabalha com memória virtual e todos os algorítmos foram desenvolvidos e implementados em FORTRAN IV, por ser uma linguagem muito conhecida tanto em Universidades quanto em empresas de uma forma geral.

O texto foi desenvolvido de forma a tornar de fácil compreensão os conceitos apresentados. Após esta introdução, apresentamos a filosofia do que é memória virtual, suas potencialidades e restrições. Em seguida analisamos como estes conceitos foram implantados no B6700. Explicações e gráficos são incluídos de forma a facilitar a exposição. Algumas considerações sobre vetores e matrizes são apresentadas sobre a ótica de estrutura de dados. Considerações sobre como as matrizes são armazenadas em FORTRAN permitem que o leitor possa avaliar os problemas relacionados com o armazenamento de grandes matrizes. Esta é a estrutura do capítulo 2, que chamamos de conceitos básicos e que são suficientes para acompanhar as elucidações fornecidas posteriormente sobre o tema central deste trabalho. Foi introduzido um terceiro capítulo para resumir todas as principais operações realizadas comumente em álgebra de matrizes, como adição, multiplicação, produto, inversão e considerações sobre tipos especiais de matrizes que geralmente aparecem em sistemas lineares. No quarto capítulo são discutidos os principais aspectos de Sistemas Lineares que interessam para o escopo deste trabalho, como os métodos de eliminação e iterativos usados para a solução de tais siste-

mas. Algumas comparações sobre os métodos são feitas em termos de aplicabilidade e de eficiência para posicionar o leitor sobre os problemas que serão tratados no capítulo seguinte. Finalmente, chegamos ao capítulo cinco onde trataremos, com o nível de detalhe considerado adequado, as matrizes esparsas, sua caracterização, alternativas de armazenamento em computador buscando uma forma eficiente para sua recuperação posterior em programas de aplicação. As principais operações com matrizes esparsas são discutidas sob o ponto de vista conceitual e em seguida apresenta-se os algorítmos que foram desenvolvidos para o B6700 do NCE/UFRJ em FORTRAN IV. Uma análise dos problemas encontrados e dos resultados observados se segue, permitindo que o leitor possa compreendê-los e adaptá-los para outras linguagens e outros equipamentos, pois durante todo o trabalho procurou-se não usar facilidades específicas do B6700, o que tornaria mais trabalhosa a tarefa de conversão dos algorítmos para outros sistemas de processamento de dados de nosso país. Tendo em vista que Sistemas de Equações Lineares se aplicam a uma diversa gama de problemas em áreas diversas e que existe um esforço no país em termos de criação e desenvolvimento de um "software" nacional, são apresentadas, não só as conclusões relativas ao trabalho, mas também algumas recomendações relativas ao tratamento de matrizes esparsas de grande porte como uma parte importante de um pacote de Programação Linear que se venha a desenvolver em nosso país. Apesar das recomendações serem feitas para sistemas que venham a ser desenvolvidos em FORTRAN IV, acreditamos que os leitores possam adaptar com pequeno esforço seus algorítmos e idéias para os recursos existentes em suas instalações.

2. CONCEITOS BÁSICOS

A compreensão das discussões sobre operações com matrizes esparsas depende de um conjunto de conhecimentos básicos que são discutidos neste capítulo. Para simplificar a apresentação, o assunto foi dividido em quatro seções distintas. A primeira delas apresenta a filosofia de memória virtual, suas potencialidades e restrições. A segunda seção descreve como esta filosofia foi implantada no B6700, tendo em vista que esta foi a máquina que se utilizou para testar os algorítmos desenvolvidos. Ela poder ser pulada sem prejuízo de compreensão. A terceira seção descreve sob o ponto de vista de estrutura de dados os vetores e matrizes de uma forma geral. A quarta seção apresenta como as matrizes são armazenadas em FORTRAN, pois esta foi a linguagem empregada para implementação dos algorítmos mencionados no texto.

2.1 MEMÓRIA VIRTUAL

Em virtude do estudo sobre Memória Virtual não se constituir no tema central deste trabalho, apresentou-se nesta seção apenas os aspectos mais relacionados com o estudo principal e detalhes adicionais podem ser encontrados em |2|.

Os módulos de gerência de memória de um sistema operacional são responsáveis pelo controle do uso da memória principal. Muitos destes módulos trabalham com a filosofia de permitir que uma tarefa seja processada apenas se houver memória suficiente disponível para carregar todos os endereços mencionados no programa. Esta restrição geralmente resulta em

áreas livres e não utilizadas na memória principal, muito embora possam haver tarefas esperando para ser carregadas e executadas. Deste modo os programadores são obrigados a construir algorítmos que se voltam mais para os aspectos de minimização do uso de memória principal do que para aqueles relacionados à rapidez de processamento. Esta pressão geralmente conduz a maiores custos de desenvolvimento e de manutenção de programas |3|.

Consideremos o caso brasileiro, onde existem muitas empresas de pequeno e médio porte que não reunem condições de ter um equipamento de grande porte em suas instalações. Por exemplo, existem centenas de equipamentos como o IBM 1130, que são ainda largamente utilizados por Universidades, empresas de engenharia, e empresas comerciais em diversas regiões do país, inclusive nos grandes centros. Equipamentos como o citado, não possuem uma grande capacidade de memória principal e se o usuário deseja desenvolver ou processar uma aplicação de um porte maior, terá certamente que fazer uso do manuseio de arquivos em disco magnético para transferir dados entre programas como uma conseqüência direta desta capacidade reduzida de memória principal. Outra alternativa é segmentar a aplicação em diversos módulos e o próprio usuário controlar a seqüência do processamento construindo uma rotina principal que irá gerenciar qual ou quais módulos devem estar residentes em um dado instante na memória principal do equipamento em questão. Em qualquer das alternativas citadas, o usuário estará construindo rotinas, pensando sempre na grande restrição que é a excassa disponibilidade de memória principal, buscando meios de superar esta restrição e con

seqüentemente recaindo em soluções que apresentam uma característica de elevado tempo de processamento, de entrada e saída e com tempos de entrega ("turn around") inaceitáveis em muito casos.

Uma solução para este problema é o emprego de máquinas com grande quantidade de memória principal. Em virtude dos elevados preços deste componente tão importante, esta alternativa pode ser considerada inviável na maioria dos casos. Uma outra alternativa que vem ganhando cada vez mais aceitação é o emprego de um sistema operacional que seja capaz de produzir a ilusão de que o computador possui uma grande memória principal. Como esta grande memória realmente não existe, ela é chamada de "memória virtual" |4|. Em sistemas de memória virtual, a memória principal, é dividida em áreas de tamanho fixo. O programa é particionado em segmentos, e cada segmento em páginas que possuem o mesmo tamanho das áreas de memória. Cada segmento pode ser definido com um agrupamento lógico de informação, tal como uma subrotina, uma matriz, um subprograma, ou uma área de dados comuns entre um programa e suas subrotinas. Deste modo, ao se carregar um programa na memória, o seu primeiro segmento é armazenado, e onde cada página ocupa uma área de memória. O módulo de gerência de memória cria algumas tabelas para calcular em tempo de execução os endereços dos ítens requeridos, que podem estar na memória principal ou na memória auxiliar. Este procedimento é realizado por uma combinação de "software" e "hardware" para que se obtenha uma eficiência maior. Nos sistemas operacionais do passado, a alocação de memória era completamente estática. A memória era alocada ou rearranjada somente quando

um programa começava ou terminava. Nos sitemas com memória virtual, é necessário alocar e desalocar a memória durante a execução do programa. Analisemos o caso em que um programa em execução necessite de dados de uma certa página. (Veja a figura 2.1). Se a página estiver na memória principal a execução pode continuar normalmente. Caso não esteja, ela terá que ser localizada, e trazida para a memória principal e a execução terá que ser suspensa até que a operação seja completada. Para carregar a página requerida ou tem de haver espaço disponível na memória principal ou alguma página carregada na memória terá que ser descarregada para disco para que se possa continuar a operação. O critério para escolha de qualquer página da memória deve ser descarregada, se for este o caso, é fundamental para evitar operações desnecessárias de entrada e saída. Um critério muito usado e também já conhecido como LRU, ou o método da página há mais tempo sem uso |2|. Quando a operação é terminada, as tabelas adequadas são atualizadas e a execução prossegue.

Em sistemas onde há gerência de demanda de páginas, pode-se observar uma interação bem próxima entre o "hardware" e o "software". Isto pode ser visto na figura 2.1. O número de tarefas executadas por "hardware" varia de um equipamento para outro. Uma discussão de performance dos métodos para descarregar páginas para memória auxiliar se encontra em |2|.

Após esta breve apresentação dos sistemas que gerenciam a demanda de memória paginada pode-se destacar as seguintes vantagens por sua utilização:

a) Grande memória virtual: um programa não está mais restrito

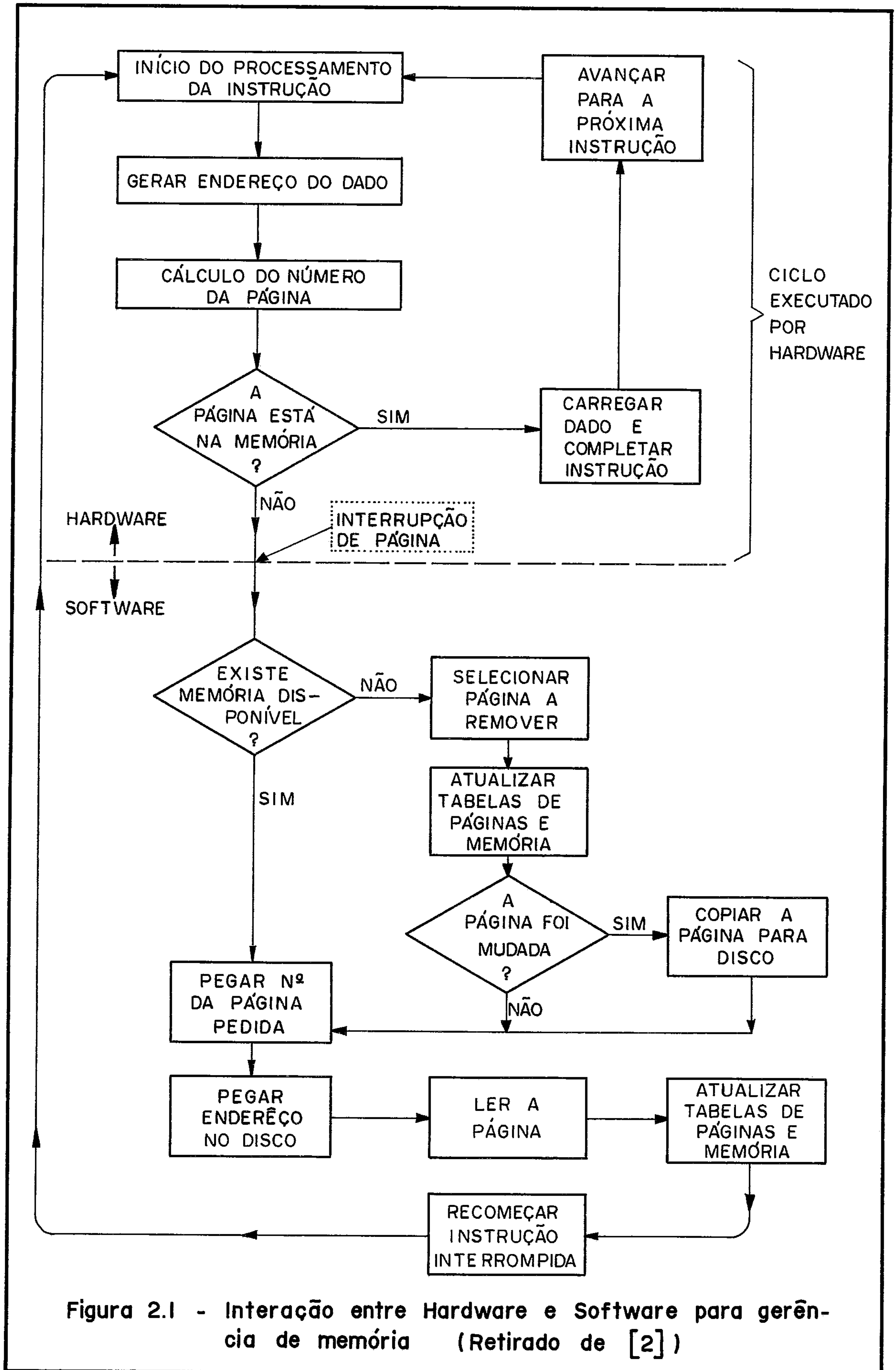


Figura 2.1 - Interação entre Hardware e Software para gerência de memória (Retirado de [2])

ao tamanho físico da memória principal. Isto permite uma compatibilização maior entre computadores pequenos e grandes. Por exemplo, um programa de 512k-bytes que normalmente é processado num computador de 1024k-bytes, pode ser processado em um computador de 256k-bytes se necessário.

b) Uso mais eficiente da memória: as partes de um programa que são raramente usadas não necessitam estar na memória principal, como por exemplo rotinas de erro que só são utilizadas em situações especiais.

c) Aumento do grau de multiprogramação: o grau de multiprogramação era limitado pelo tamanho da memória principal em outros sistemas de gerência de memória. Isto é, a soma dos tamanhos dos programas deveria ser menor ou igual ao tamanho da memória principal. Nos sistemas que gerenciam a demanda de memória paginada, não existe tal limitação. Estes sistemas permitem um maior número de tarefas sendo multiprogramadas em um computador com memória principal pequena.

Se alguns problemas foram resolvidos, igualmente algumas desvantagens foram observadas nos sistemas de memória virtual:

a) Aumento do número de tabelas e maior sobrecarga no processador para gerenciar o carregamento e descarregamento de páginas, quando comparados aos sistemas tradicionais.

b) Como conseqüência da falta de limitação explícita no tamanho dos programas ou no grau de multiprogramação, mecanismos especiais devem ser desenvolvidos para evitar as situações de "thrashing", como numa situação extrema onde 90 por cento do tempo do processador fosse consumido para manusear

interrupções de demanda de páginas, enquanto menos de 10 por cento do tempo fosse utilizado pelos programas de aplicação.

O conhecimento destas desvantagens será de grande valia para nosso estudo, pois o armazenamento e manipulação de matrizes esparsas de forma inadequada conduzirão a um agravamento das desvantagens citadas.

2.2 MEMÓRIA VIRTUAL NO B6700

A série B7000/B6000 de computadores da Burroughs utilizam uma técnica de memória virtual baseada em segmentos de programas de tamanhos variáveis. A segmentação dos programas é executada pelos próprios compiladores. No caso do FORTRAN a segmentação é feita a nível de subrotinas.

Segmentos de programas de tamanhos variáveis otimizam o uso da memória principal do seguinte modo:

a) Eliminando segmentos desnecessários da memória principal.

b) Requisitando apenas a memória necessária para armazenar o segmento a ser usado e não como em outras técnicas onde é necessário alocar uma área de memória grande e que seja suficiente para armazenar o maior dos segmentos do programa.

Nos sistemas de segmentos de programa de tamanhos variáveis, os segmentos estão residentes somente quando são requisitados para uma tarefa.

Para cada segmento de programa existe um segmento descritor associado. Os segmentos do programa e um dicionário dos segmentos do programa são armazenados em disco. A área de dados também possuí um descritor associado com cada

matriz de dados.

Os descritores referenciam cada área da memória virtual que estão em uso e aquelas áreas da memória auxiliar que contêm segmentos de programas ou de dados relacionados com uma tarefa em execução. Os descritores possuem informação sobre o tipo de área (de programa ou de dados), o tamanho da área, e o endereço da área na memória ou em disco. Em cada descritor existe também um bit de presença. Este bit indica a presença ou ausência (na memória) de um segmento de programa de dados. Quando o bit está ligado o segmento está na memória. Quando o bit está apagado o segmento está na memória auxiliar.

Em determinados casos, um descritor pode conter informações sobre outros descritores. Consideremos o caso de uma matriz de duas dimensões em FORTRAN. Sua representação na memória é feita através de um descritor que aponta para uma área que contêm outros descritores de dados; cada qual representando um valor que é um índice da primeira dimensão da matriz. Tais áreas são chamadas de "dope vectors". Veja a figura 2.2.

Cada elemento do "dope vector" descreve uma linha da matriz. Cada elemento de um "dope vector" pode descrever outro "dope vector". Isto é fundamental para permitir a descrição de matrizes multi-dimensionais.

Grande área de dados (excedendo a 4095 palavras) são automaticamente segmentadas em segmentos de 256 palavras a fim de evitar uma grande demanda ao sistema por excessivas áreas contíguas de memória principal. Áreas segmenta

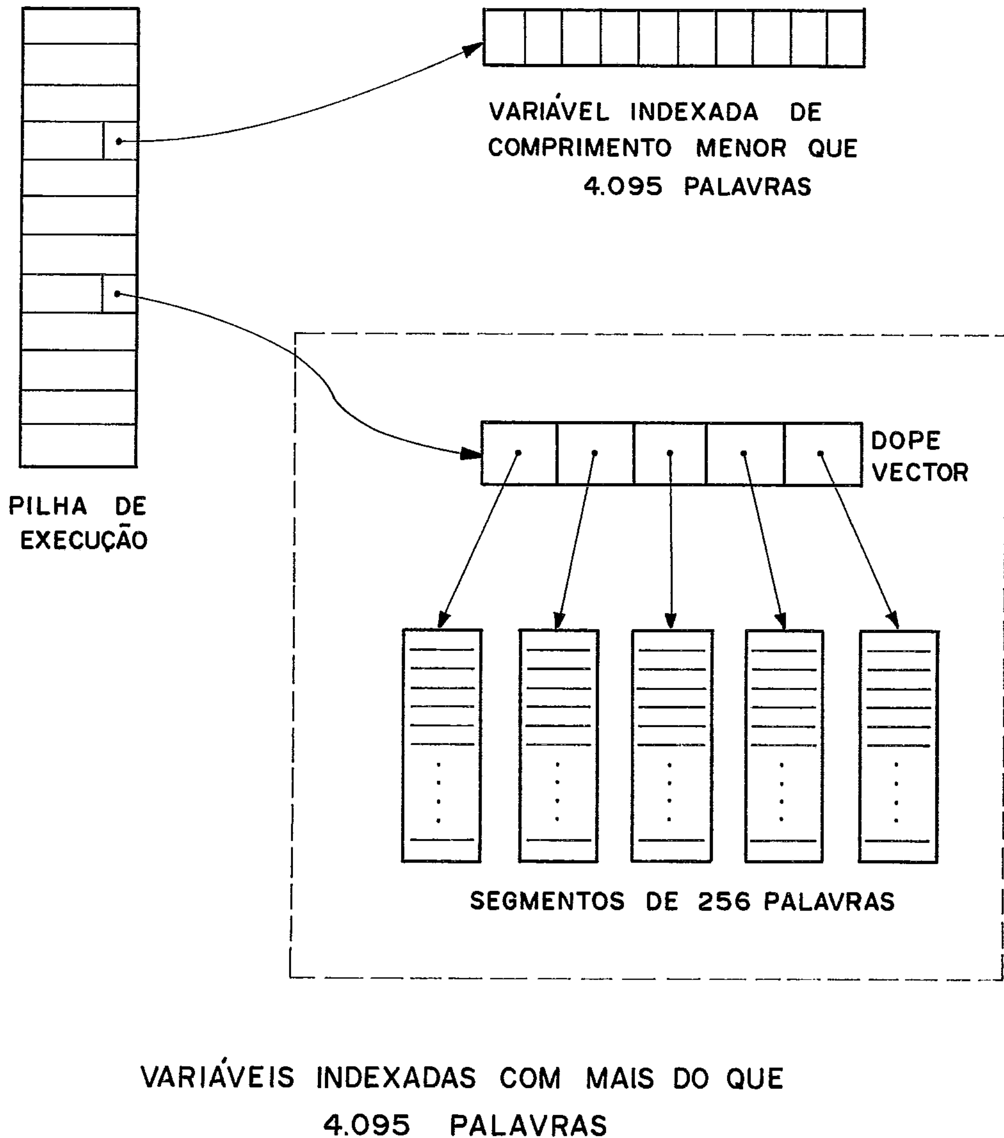


Figura 2.2 - Estrutura de matrizes no B 6700
(Retirado de [13])

das deste modo são indicadas como tal, através de um bit de segmentação ligado em seu descritor.

Um descritor de dados referencia uma área de dados, inclusive áreas intermediárias de entrada e saída. O descritor de dados define uma área de memória começando em um endereço base contido no descritor. O tamanho da área de memória em operandos é contido no campo de comprimento do descritor. Descritores de dados podem referenciar qualquer palavra de memória de 0 até 1.048.576. A estrutura do descritor de dados pode ser vista na figura 2.3, e contém o seguinte:

1. Bits 50,49 e 48, possuem sempre o valor 101 armazenado.
2. Bit 47, o bit de presença, indica a presença ou ausência do dado na memória principal. Um 0 causa uma interrupção sempre que o descritor é usado por um processador para conseguir dados que não estejam presentes. Um 1 indica que o dado descrito está na memória principal.
3. Bit 46, o bit de cópia. Um 0 indica que este é o descritor original para a área de dados específica. Um 1 indica que este descritor é uma cópia do descritor original.
4. Bit 45, o bit de indexação. Um 0 indica que uma operação de indexação é necessária para que o descritor possa ser usado para obter um dado. Um 1 indica que a indexação já ocorreu e que o valor do índice está armazenado no campo de comprimento ou índice.
5. Bit 44, é o bit a segmentação. Um 0 indica que os dados não estão segmentados. Um 1 indica que os dados estão divididos em segmentos.

								CAMPO DE COMPRIMENTO / ÍNDICE						ENDERÊÇO DE MEMÓRIA / DISCO				
	47 P		43 R					39	35	31	27	23		19	15	11	7	3
50 I		46 C		42 Ø				38	34	30	26	22		18	14	10	6	2
49 Ø			45 I		41 Ø			37	33	29	25	21		17	13	9	5	1
48 I				44 S		40 D		36	32	28	24	20		16	12	8	4	Ø

Figura 2.3 - Estrutura de um descritor de dados (Retirado de [7])

6. Bit 43, o bit que indica "read-only". Um 0 indica que os dados podem ser referenciados para leitura e gravação. Um 1 indica que a área não pode ser usada para armazenamento.
7. Bits 42 e 41, um 0 indica que esta é uma palavra para descrição de dados.
8. Bit 40, o bit da dupla-precisão. Um 0 indica operandos de precisão-simples, um 1 indica operandos de dupla-precisão.
9. Bits de 39 a 20, contêm tanto o comprimento da área de memória (se o bit 45 estiver desligado) ou um índice (se bit 45 estiver ligado). Se o bit 45 vale 0, o descritor não foi indexado. Este campo é usado para validação de comprimento durante a indexação. Se o bit 45 vale 1, o descritor foi indexado. Para uma operação de precisão dupla, o índice é multiplicado por dois após a verificação de comprimento, e o resultado é armazenado no campo de índice.
10. Bits de 19 a 0, contêm tanto um endereço de memória principal ou de disco. Se o bit de presença (bit 47) vale 1, este campo contém o endereço de memória do dado. Se o bit de presença vale 0 e o bit de cópia (bit 46) vale 0, este campo contém o endereço do dado em disco. Se o bit de presença vale 0 e o bit de cópia vale 1, este campo contém o endereço de memória do descritor original.

O sistema operacional estrutura a memória de tal modo que cada área é precedida e seguida por palavras protegidas de memória. Estas palavras protegidas possibilitam a estrutura ligada ou lista usada para a gerência de memória. Dois tipos de listas são mantidas pelo sistema. Uma lista

contêm apontadores para as áreas em uso e a outra lista contêm apontadores para as áreas disponíveis.

As áreas em uso podem ser de dois tipos: aquelas que devem estar residentes desde a hora em que foram alocadas até que chegue o momento de sua desalocação; o outro tipo é daquelas áreas que podem ser descarregadas para disco pelo sistema operacional, desde que possam ser chamadas automaticamente quando necessário.

Pedidos de novas áreas de memória ocorrem quando uma tarefa referencia um descritor de programa ou de dados que possua o bit de presença desligado. Tal referência causa uma interrupção e, através de uma interação entre hardware e software, o pedido será atendido, se possível. Quando um pedido não pode ser atendido por não haver espaço disponível na lista de área livres, o sistema procura realizar um "overlay" para conseguir a área requisitada.

No B6700 um processo se caracteriza por um Segmento de Dados e dois outros chamados Dicionário de Segmentos do Sistema e Dicionário de Segmentos do Processo.

O Segmento de Dados ou pilha de execução é usado para armazenar variáveis locais ao processo, parâmetros de procedimentos, apontadores para descritores de procedimentos ou dados como vetores e matrizes localizados fora da pilha de execução.

O Dicionário de Segmentos do Sistema ou tronco de pilha é compartilhado por todos os processos do sistema.

O Dicionário de Segmentos do Processo é em ge-

ral particular ao processo.

O relacionamento entre os segmentos mencionados (vide figura 2.4) permite que em qualquer instante, partindo-se de um endereço lógico e usando-se os registradores D0, D1, e D2 possamos ter acesso às informações necessárias para a execução.

Como principais características de esquema de memória virtual no B6700 podemos citar as seguintes:

1. Ele manipula área de dados de comprimentos variáveis.
2. A memória é estruturada até o nível de matrizes em um programa e proteção de memória por "hardware" é aplicada até aqueles níveis.
3. Para extensão da memória são usados sistemas de discos fixos de acesso rápido (um cabeçote por trilha).

As principais características do módulo de gerência de memória são as seguintes:

1. Praticamente toda a memória é endereçável através de descritores que possuem um endereço base e um campo de comprimento.
2. Os descritores possuem um bit de presença que causa uma interrupção de "hardware" caso a área referenciada não esteja presente na memória principal quando o descritor é consultado.
3. O campo de endereço base contém um endereço absoluto de memória se o dado está presente ou um endereço relativo em disco se está ausente.

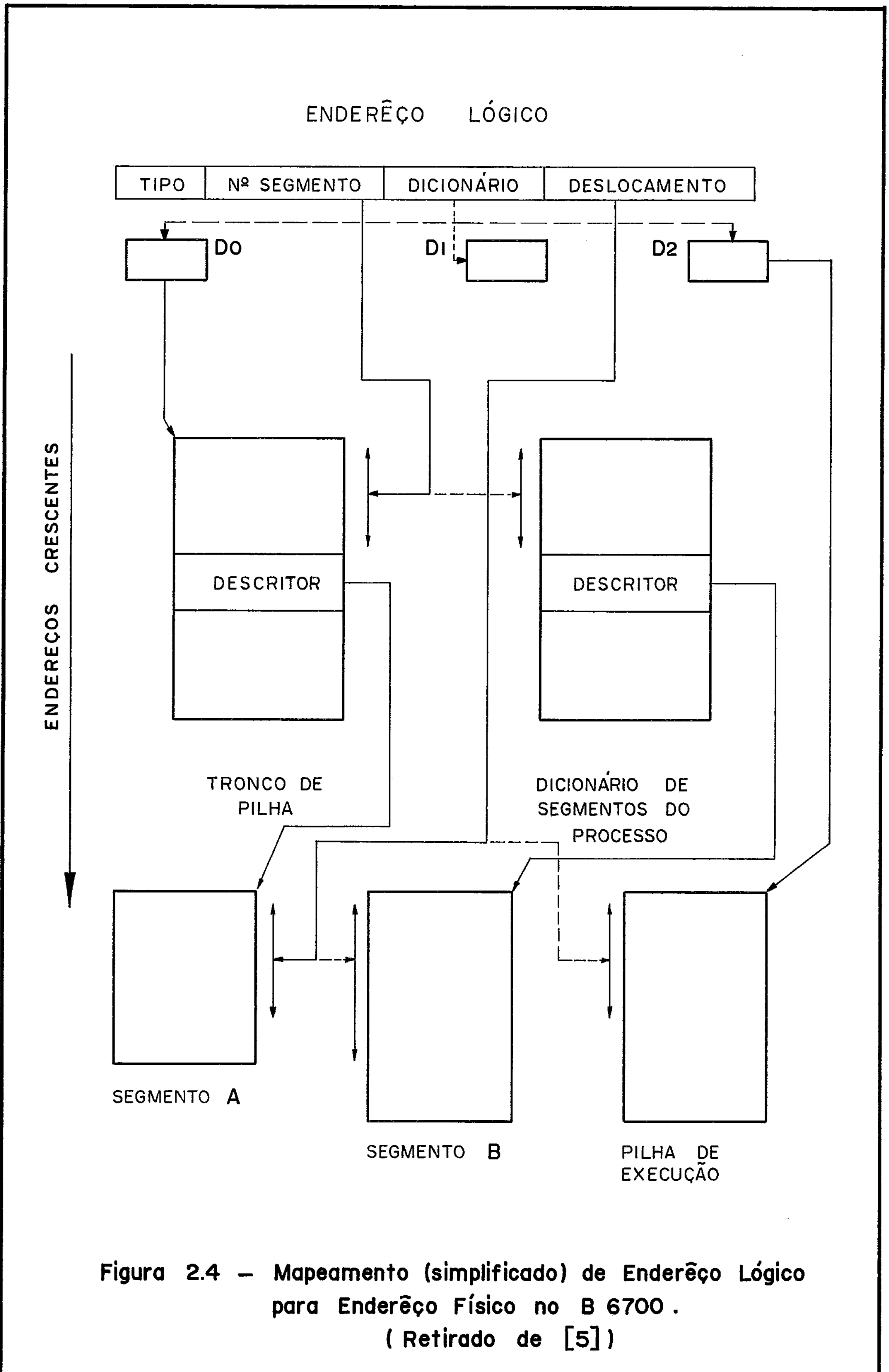


Figura 2.4 — Mapeamento (simplificado) de Enderêço Lógico para Enderêço Físico no B 6700. (Retirado de [5])

4. Palavras específicas de memória são alcançadas através de indexação do descritor de dados em uso. Se o índice é negativo ou maior que o campo de comprimento do descritor de dados, uma interrupção por índice inválido acontecerá. Deste modo a memória é protegida de erros cometidos pelo próprio usuário ou de outros.

5. A memória é organizada em área em uso e disponíveis, ambas sendo estruturadas através de ligações formando listas para facilitar a busca. As ligações das áreas em uso apontam de volta para o descritor que a referencia.

6. Áreas em uso são organizadas em áreas que podem ou não sofrer um operação de "overlay". Na maioria dos casos, os programadores não podem definir áreas fixas que não sofram "overlay".

7. Quando uma área de memória é requerida, inicialmente a lista de área disponível é consultada. Se o pedido não pode ser atendido a partir desta lista, então a lista de áreas em uso deve ser analizada e um "overlay" se torna necessário.

8. Quando dados devem sofrer um "overlay", a versão atual deve ser descarregada para disco para preservar a informação. Áreas de programas em linguagem máquina podem passar por um "overlay" facilmente pois já existe uma cópia fiel armazenada em disco.

Memória Virtual oferece ao programador uma área quase ilimitada de memória através da grande capacidade dos dispositivos de memória auxiliar. No entanto, o programador deve considerar que em sua instalação a quantidade de me-

mória principal é limitada e que é sua responsabilidade exercer um rígido controle sobre as quantidades de memória necessárias ao seu programa.

Aliando-se esta característica de alocação dinâmica de memória, possibilidades inerentes aos programas em FORTRAN podem ser usadas para otimizar o uso de memória quando necessário e na maior parte das aplicações rotineiras podem ser simplesmente esquecidas.

Um ponto fundamental no entanto, é que Memória Virtual não absolve o programador de sua responsabilidade de controlar o uso de memória principal.

2.3 VETORES E MATRIZES

Quando mencionamos a palavra vetor em aplicações das mais variadas temos em mente uma lista ordenada de valores, chamados componentes, que são interpretados como coordenadas em um espaço multi-dimensional. A partir deste conceito básico, decorre o conceito de matriz, um vetor cujos componentes são vetores. Outra maneira de caracterizar os vetores, seria considerá-los como conjuntos ordenados formando uma lista, cujos componentes seriam todos do mesmo tipo. Deste modo uma lista de cinco cadeias alfabéticas será um vetor, e o valor do terceiro componente, considerado como uma variável, será a terceira cadeia alfabética. A idéia de ordem no conjunto, se relaciona com uma correspondência aos números inteiros positivos, 1, 2, ..., e deste modo pode-se definir um vetor como um mapeamento dos inteiros a um conjunto de compo-

nentes, todos do mesmo tipo |9|.

Para armazenarmos vetores e matrizes na memória de um computador necessitamos conhecer as suas dimensões. A dimensão de um vetor é um número inteiro, igual ao número de componentes. O conhecimento antecipado da dimensão de um vetor é muito importante para que se possa construir uma função de mapeamento que permita acesso eficiente e para que se possa reservar uma área para todos os componentes. Outra implicação de conhecer a dimensão, é que uma operação definida para um par de componentes pode ser extendida a um par de vetores que possuam as mesmas dimensões. No caso em que a dimensão de um vetor não é conhecida até a hora da execução, uma função de mapeamento baseada em listas pode ser mais eficiente. No entanto, mesmo neste caso, uma função de mapeamento indexada pode ser usada, se um valor superestimado para a dimensão puder ser fornecido. Podemos dizer também que um vetor é uma matriz de uma dimensão.

A partir destes conceitos, uma matriz de n dimensões pode ser considerada como um conjunto de componentes identificadas por um conjunto ordenado de índices $(i_1, i_2, \ldots, i_n)$, onde $1 \leqslant i_j \quad m_j$, $j=1, 2, \ldots, n$. Cada elemento da matriz será representado por $d(i_1, i_2, \ldots, i_n)$. O conjunto ordenado $(m_1, m_2, \ldots, m_n)$ é a lista de dimensões da matriz e o número de componentes para a mesma pode ser expresso por

$$\prod_{i=1}^{n} m_i$$

Exatamente como no caso dos vetores, o conhecimento antecipado da lista de dimensões pode ser de muita vantagem para a de

finição de funções de mapeamento que permitam uma recuperação de modo eficiente em algorītmos diversos.

Outro modo de conceituarmos as matrizes pode ser visto em |10| onde a generalização de uma lista linear ē discutida. A matriz A de dimensões m por n, por exemplo, pode ser representada por

$$\begin{bmatrix} A(1,1) & A(1,2) & \dots & A(1,n) \\ A(2,1) & A(2,2) & \dots & A(2,n) \\ \dots & \dots & \dots & \dots \\ A(m,1) & A(m,2) & \dots & A(m,n) \end{bmatrix}$$

Nesta matriz bidimensional, cada elemento A(j,k) pertence a duas listas: a lista da linha j composta por A(j,1), A(j,2) , ..., A(j,n), e a lista da coluna k composta por A(1,k), A(2,k), ..., A(m,k). As listas ortogonais de linhas e colunas formam a matriz. Esta abordagem pode ser extendida igualmente para matrizes multi-dimensionais.

As matrizes bidimensionais, apesar de sua forma retangular e da similaridade com a espēcie de notação que usualmente se emprega em matemātica, são armazenadas na memōria de um computador de outros modos. Uma sērie de memōria ē reservada a partir da lista de dimensões e cada componente da matriz ē armazenado sequencialmente, em posições contīguas. Esta alocação de memōria ē chamada de alocação seqüencial. Este ē o caso mais comum de armazenamento de matrizes.

Um procedimento extensamente usado para armazenar matrizes ē aquele chamado de "armazenamento em ordem lexicogrāfica", ou seja armazenamento por linhas. Para uma ma-

triz A de quatro dimensões, onde cada elemento será referenciado por A(i, j, k, l) e também $0 \leq i \leq 2$, $0 \leq j \leq 4$, $0 \leq k \leq 10$, $0 \leq l \leq 2$, teríamos a seguinte seqüência de armazenamento:

A(0,0,0,0), A(0,0,0,1), A(0,0,0,2), A(0,0,1,0), A(0,0,1,1), ..., A(0,0,10,2), A(0,1,0,0), ..., A(0,4,10,2), A(1,0,0,0), ..., A(2,4,10,2).

Neste caso a matriz é armazenada em posições seqüenciais de memória, com o valor do último subscrito variando mais rapidamente que os subscritos anteriores, de modo a percorrer todas as linhas da matriz. No exemplo apresentado, a matriz A tem seus elementos A(i, j, k, l) dispostos sequencialmente com o último subscrito l variando inicialmente de 0 até 2, e todos os demais com o valor zero. Em seguida o subscrito k passa a ser incrementado de 1 e o subscrito l varia novamente de 0 até 2. Quando o subscrito já assumiu todos os seus possíveis valores, é chegada a vez do subscrito j variar. Em resumo os primeiros subscritos variam mais lentamente que os últimos subscritos. Fixado o valor de um subscrito mais à esquerda, todos os subscritos mais à direita variam mais rapidamente, do seu valor inicial até o seu valor final e em seguida o processo recomeça até que todos os subscritos hajam assumido todos os seus possíveis valores.

A fórmula geral de recuperação de qualquer elemento da matriz $A(i_1, i_2, \ldots, i_k)$ assim armazenada e onde cada elemento mede c palavras e cuja lista de dimensões seja $0 \leq i_1 \leq d_1$, $0 \leq i_2 \leq d_2$, $0 \leq i_3 \leq d_3$, ..., $0 \leq i_k \leq d_k$ conforme definido em |10| é a seguinte:

$$LOC\ (A(I_1, I_2, \ldots, I_k)) = LOC\ (A(0, 0, \ldots, 0)) +$$
$$c(d_2+1)\ldots(d_k+1)I_1 + \ldots +$$
$$c(d_k+1)I_{k-1} + cI_k =$$
$$= LOC\ (A(0, 0, \ldots, 0) + \sum_{1 \leq r \leq k} a_r I_r \quad \text{onde}$$

$$a_r = c \prod_{r < s \leq k} (d_s+1)$$

O método apresentado pelo citado autor é apropriado para armazenar matrizes retangulares, e quando a maioria de seus elementos são diferentes de zero. Casos especiais de matrizes não serão analisados nesta seção.

Outra maneira similar de armazenamento seria aquela por colunas, ao invés de linhas, como foi apresentado no texto. Geralmente, prefere-se trabalhar com matrizes, na maioria dos algorítmos, por linhas. Porém devemos ressaltar que muitos compiladores armazenam as matrizes por colunas, como é o caso do FORTRAN.

2.4 ARMAZENAMENTO DE MATRIZES EM FORTRAN

Após a discussão sobre vetores e matrizes da seção anterior, passaremos agora a analisar como as matrizes são armazenadas em FORTRAN IV. No final desta seção serão mencionados alguns detalhes específicos do FORTRAN do computador B6700.

Em FORTRAN, as variáveis são divididas em variá-

veis simples e indexadas. As variáveis indexadas são denotadas por um nome de variável seguido por uma lista de subscritos entre parênteses. Uma variável indexada é um conjunto ordenado, correspondendo a uma organização de n dimensões de tal modo que cada membro possa ser referenciado como um elemento do conjunto com cada um dos n subscritos do elemento relacionando-se com sua localização nas dimensões apropriadas.

Uma variável indexada é referenciada por um identificador, que é o nome do conjunto, que deve ter a mesma forma de um nome de variável simples. Este identificador representa os dados armazenados no conjunto, os quais são todos do mesmo tipo. O tipo é indicado pelo nome da variável, seguindo as mesmas regras para definição de tipo que são utilizadas pelas variáveis simples.

De uma forma compacta podemos apresentar os membros do conjunto ou elementos assim:

a (s)

onde a é um nome de variável simples, formado de acordo com as regras que governam tais construções, e s é a lista de subscritos que consiste de tantas expressões aritméticas (subscritos) quantos forem as dimensões do conjunto, separadas por vírgulas.

Um nome de variável é designado como uma variável indexada através de comandos de especificação como DIMENSION, EQUIVALENCE, EXTERNAL, IMPLICIT, COMMON, COMPLEX, DOUBLE PRECISION, REAL*8, INTEGER, LOGICAL ou REAL. Estes comandos de especificação mostrados, são utilizados para fornecer informações em tempo de compilação sobre variáveis de um progra

ma tais como: tipo da variável, valores iniciais, alocação de memória e para permitir que subprogramas ou subrotinas possam utilizá-las como parâmetros. Estes comandos de especificação apresentados servem para declarar o número máximo de dimensões permitido para a variável e devem preceder a primeira utilização do nome da variável, tanto num comando executável quanto num DATA.

Em um segmento de programa, um identificador pode ser usado como variável simples ou indexada, mas nunca nos dois casos ao mesmo tempo. Sempre que um nome de variável indexada aparecer em um programa, deve ser imediatamente seguido por uma lista de subscritos, exceto quando o nome da variável indexada estiver em:

a) Uma lista de argumentos fantasmas de um subprograma.

b) Uma lista de argumentos numa chamada a um subprograma.

c) Numa lista de variáveis de um comando de entrada ou saída.

d) Um COMMON, DATA, EQUIVALENCE, ou especificação de tipo.

e) Um comando READ, WRITE, PRINT, ou PUNCH, substituindo o número do FORMAT associado.

Maiores definições e detalhes sobre variáveis indexadas podem ser encontradas em |13|.

As variáveis indexadas em FORTRAN permitem que o usuário possa organizar áreas de memória de uma forma conveniente para seus algorítmos,matemáticos principalmente. Internamente, no entanto, uma variável indexada é armazenada como um grupo de uma ou mais palavras contíguas de memória. Uma variável indexada em FORTRAN, com qualquer número de dimen-

sões declaradas, é representada internamente como uma matriz de uma dimensão (ou vetor).

Neste trabalho vamos restringir nossa discussão a matrizes bi-dimensionais de tipo real. Cada elemento da matriz referenciada, requer a mesma quantidade de memória que uma variável simples do mesmo tipo da variável indexada. Se por acaso, for necessário trabalhar com matrizes de precisão extendida, definidas por DOUBLE PRECISION ou REAL*8, basta considerar que cada elemento ocupará, por exemplo, o dobro de memória, se no computador em questão isto for válido.

Para variáveis indexadas de apenas uma dimensão, cada membro armazenado corresponde um elemento do vetor na mesma seqüência respectivamente.

Variáveis indexadas de mais de uma dimensão em FORTRAN, são armazenadas por colunas em um vetor contínuo de palavras de memória.

Este processo de armazenamento pode ser compreendido conforme mostrado no exemplo apresentado a seguir:

$$\begin{bmatrix} A(1,1) & A(1,2) \\ A(2,1) & A(2,2) \\ A(3,1) & A(3,2) \end{bmatrix}$$

Os elementos A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2) serão armazenados nesta ordem. Cada coluna, de cima para baixo, da esquerda para a direita, são colocadas seqüencialmente na área contígua de memória reservada para seu armazenamento.

O mesmo procedimento acontece para matrizes de n dimensões. A ordem apropriada é obtida quando se listam todos os elementos de qualquer matriz, fazendo-se que o primeiro subscrito varie mais rapidamente que os demais, em seguida repetindo-se a operação para o segundo subscrito, até que se chegue ao último.

De uma forma geral, uma matriz A de N dimensões pode ser declarada por um especificador de variáveis indexadas como por exemplo DIMENSION A(D1, D2, ..., DN) onde D1, D2, ..., DN são os valores máximos em cada dimensão. Cada elemento de A pode ser referenciado por A(I1, I2, ..., IN) desde que os valores de I1, I2, ..., IN sejam definidos. Suponhamos que cada elemento ocupe uma posição de memória no raciocínio seguinte, onde a cada elemento da variável indexada A corresponda uma posição na memória. A fórmula para se localizar a posição relativa de cada elemento na memória é a seguinte:

$$I = I1 + D1^{*}(I2-1) + D1^{*}D2^{*}(I3-1) + \ldots$$

$$\ldots + D1^{*}D2^{*}D3^{*} \ldots {}^{*}(DN-1) {}^{*} (IN-1)$$

onde I é o índice de localização variando de 1 até (D1*D2*D3* ...*DN) inclusive. Esta fórmula apresentada em |13| é chamada de "função do elemento sucessor da variável indexada".

Como este trabalho foi desenvolvido num computador Burroughs B6700, cabe ressaltar aqui alguns detalhes para melhor compreensão futura de alguns tópicos.

O B6700 possuí uma palavra de memória de comprimento de 52 bits, porém apenas 48 são acessíveis aos usuários. Os 52 bits são numerados de 51 até 0, da esquerda para

a direita. A parte da palavra que o usuário pode utilizar vai do bit 47 ao bit 0. Os bits de 51 até 48 são utilizados pelo próprio sistema.

No FORTRAN do B6700 cada variável indexada pode ter até 31 dimensões. No caso da matriz ser do tipo INTEGER, cada elemento pode ter um sinal (+ ou -) e um valor que não seja maior que 549755813887. No caso da matriz ser do tipo REAL, os valores dos seus elementos devem ter no máximo 11 dígitos significativos e um ponto que deve ocorrer antes, depois ou entre os dígitos decimais. O valor absoluto máximo de um elemento real é aproximadamente 4.3135914667E68 que é igual a ((8**13)-1)*8**63 e o menor valor absoluto diferente de zero é aproximadamente 8.7581154021E-47 que é igual a 8**(-51).

Cada elemento de uma matriz, em precisão simples, ocupa uma palavra de memória no B6700.

Um detalhe que não se deve esquecer quando trabalhamos com matrizes no B6700, é que nenhuma variável indexada pode ocupar mais do que 65.535 palavras de memória. Caso tenhamos de trabalhar com uma matriz de tamanho ainda maior, teremos que dividí-la em duas ou mais matrizes menores. Outra hipótese é não utilizar as facilidades oferecidas pelo FORTRAN para tratar variáveis indexadas e construir algorítmos para trabalhar com arquivos em memória auxiliar. Em nosso trabalho restringiremos a discussão a matrizes de no máximo 65.545 palavras de memória. Isto dá margem a tratar, por exemplo, matrizes de até 256 linhas por 255 colunas.

Como vimos, as matrizes são armazenadas em posições consecutivas de memória. Porém, no caso em que as matri-

zes sejam superiores a 4095 palavras, elas serão segmentadas para que o sistema possa realizar operações de "overlay". Cada segmento da matriz conterá, no máximo, 256 palavras e serão automaticamente carregadas e descarregadas, durante a execução de um programa.

O compilador FORTRAN do B6700 possui uma opção de compilação chamada LONG que permite ao usuário alterar a maneira como o compilador trata as matrizes. Matrizes declaradas em partes de um programa fonte onde a opção LONG esteja válida, não serão segmentadas não importando qual seja o seu tamanho. A única exceção é o tamanho máximo de matrizes que o FORTRAN pode manipular, que já foi definido anteriormente.

Maiores informações sobre o procedimento de segmentação empregado pelo compilador para matrizes e programas podem ser encontradas em |13|. Sempre que nos referenciarmos futuramente a algum tópico relacionado com os algorítmos implementados no B6700 e não tenha sido coberto nesta seção, faremos novas referências e menções específicas. Nos limitamos a mencionar aqui, apenas aquelas informações que são mais necessárias à compreensão dos algorítmos descritos no capítulo cinco.

Como principais conclusões, a partir das considerações apresentadas neste capítulo, podemos citar as seguintes:

a) Para equipamentos que possuam sistemas operacionais que funcionem com memória virtual, como o B6700, é fundamental que se construam algorítmos que minimizem a necessidade de acesso a segmentos

ou páginas que não estejam residentes, a fim de que operações de entrada e saída não sejam realizadas desnecessariamente pelo sistema operacional em tarefas de "overlay".

b) Com relação a algorítmos para manipulação de grandes matrizes deve-se deduzir que, variações em subscritos de colunas causam necessidade de páginas ou segmentos com maior freqüência do que variações de subscritos de linhas.

c) Algorítmos desenvolvidos em FORTRAN, que manipulem grandes matrizes, têm a característica de que a demanda por páginas ou segmentos não residentes, aumenta em função da primeira dimensão da matriz.

d) Para minimizar a demanda por páginas e interrupções causadas por páginas ou segmentos necessários e não residentes, é interessante que o número e a amplitude das variações nos subscritos das matrizes sejam os menores possíveis. Desta maneira, os algorítmos deveriam trabalhar coluna a coluna, em ordem natural, e as operações em cada coluna só deveriam começar, após o término de todas as operações na coluna imediatamente anterior.

e) Tendo em vista o modo de armazenamento de matrizes que é utilizado pelo compilador FORTRAN, para o tratamento de grandes matrizes dever-se-ia evitar situações onde no programa houvesse um dimensionamento exagerado e que não seja efetivamente utilizado. Por exemplo, em um programa principal existe um comando do seguinte tipo: DIMENSION A(100, 100), porém ao se efetuar a leitura da matriz verificou-se que a mesma possuía uma dimensão de 60 por 60. Isto significa que apesar da matriz necessitar de 3600 elementos para o

caso, uma área de 10000 elementos foi efetivamente alocada. Se, no entanto, não for possível ou desejável esta alternativa, poder-se-ia pensar em otimização de tempo, utilizando algorítmos de compressão de memória e de localização de elementos como aqueles das subrotinas ARRAY e LOC que fazem parte do conjunto de subrotinas do IBM Scientific Subroutine Package. Na subrotina ARRAY é executado um procedimento que comprime todos os elementos realmente utilizados numa matriz para o início da área reservada para seu armazenamento a partir do DIMENSION correspondente. No exemplo dado, os 3600 elementos da matriz seriam armazenados nas primeiras 3600 posições da área reservada para 10000 elementos. A localização de qualquer elemento A(i,j) seria feita através da subrotina LOC. A partir desta organização, todos os algorítmos para tratamento das matrizes utilizariam apenas um vetor de 3600 elementos.

Programas que manipulam matrizes em matrizes de memória virtual têm sido apresentados, com alguma freqüência, como produto de simples alterações de programas desenvolvidos para trabalhar somente com a memória principal através de pequenas mudanças na ordem de colocação de comandos DO e seus ciclos associados |12|. Passaremos em seguida a analisar as principais operações com matrizes e tentar formular estas operações ao nível matemático utilizando as conclusões apresentadas neste capítulo.

3. ÁLGEBRA DE MATRIZES

O objetivo deste capítulo é o de definir alguns termos que serão utilizados em todo o trabalho que se segue, discutir as principais operações realizadas com matrizes e, utilizando as conclusões do capítulo anterior, construir algorítmos que sejam apropriados para tratar grandes matrizes em sistemas que trabalham com memória virtual. Para analizarmos os algorítmos, algumas hipóteses foram assumidas, exceto quando especificado em contrário. Propositadamente, deixamos para o capítulo cinco o tratamento de matrizes esparsas, onde o assunto será apreciado com maior profundidade.

Para que se possa comparar algorítmos e para reduzir a demanda por páginas, suponhamos que cada matriz seja armazenada no início de uma página e que uma página contenha um número inteiro de colunas da matriz. Em aplicações de problemas reais, dependendo das dimensões da matriz, as colunas das matrizes podem começar e terminar em qualquer parte de páginas alocadas para seu armazenamento sem que o usuário possa exercer qualquer controle sobre o fato, porém os algorítmos funcionarão corretamente, podendo ser aprimorados para situações específicas. Estes refinamentos, no entanto, não alterarão a ordem de grandeza da demanda de páginas requeridas pelos mesmos. Para simplificar a apresentação dos algorítmos não incorporamos estas características de otimização, assim como não nos preocupamos em mostrar rotinas de erros, tratamento e "overflows" e de "underflows", e impressão de avisos em casos de mal funcionamento dos algorítmos como conseqüência de erros de dados. Para utilização de tais algorítmos em aplica-

ções, tais cuidados deveriam ser tomados pelo usuário.

Os cálculo de demanda de páginas apresentados para os algorítmos foram feitos para uma matriz definida por DIMENSION A(128, 128) em um sistema hipotético onde sua página mede 1024 palavras. Nas fórmulas apresentadas, n é a ordem da matriz considerada, d é o valor da primeira dimensão, w é o número de palavras contidas em uma página da memória virtual, k é o número de colunas da matriz contidas em uma página (valor inteiro por hipótese), e p é o número de páginas que a matriz ocupa no total.

3.1 DEFINIÇÕES

O conceito de matriz tem sido usado desde o século XIX. Em 1858 um algebrista chamado Arthur Cayley publicou um trabalho que chamou "Memória sobre a teoria de matrizes" e que modificou segundo |18| muitos dos conceitos estabelecidos até àquela época. Hoje estes conceitos são largamente utilizados em engenharia, química e física, estatística e programação matemática.

Uma matriz de m por n ou (m,n) é um conjunto retangular de elementos arranjados em m linhas e n colunas, podendo ser representada assim:

$$A = |a_{ij}| = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Dizemos que duas matrizes são do mesmo tamanho se possuírem o mesmo número de linhas e de colunas.

Duas matrizes são iguais se e somente se elas forem idênticas; isto é, se e somente se tiverem o mesmo número de linhas e de colunas, e se todos os seus elementos de mesma posição forem iguais.

Uma matriz formada por apenas uma linha é chamada de matriz linha. Uma matriz formada de apenas uma coluna é chamada de matriz coluna. Ambas são geralmente chamadas de vetores.

Uma matriz que possua igual número de linhas e de colunas é chamada de matriz quadrada.

Para uma matriz $A = (a_{ij})$ de m linhas e de n colunas, onde temos $i = 1,2,3,\ldots,m$ e também $j = 1,2,3,\ldots,n$ a matriz representada por $A^T = (a_{ji})$ de n linhas e de m colunas é também chamada de matriz transposta.

A seqüência de elementos $a_{11}, a_{22}, a_{33}, \ldots, a_{nn}$ de uma matriz quadrada é chamada de diagonal principal.

Por definição, um determinante é uma função definida sobre uma matriz quadrada, calculada de acordo com um conjunto de regras para combinar os elementos da matriz. O determinante de uma matriz A é representado por det A ou também por $|A|$. O determinante de uma matriz é um número escalar e não deve ser confundido com a própria matriz A. A regra utilizada para combinar os elementos para o cálculo é a seguinte:

O determinante de uma matriz quadrada de dimensão n por n é obtido pela formação da soma dos produtos de to

das as possíveis combinações de elementos tomados $\underline{n}$ de cada vez, um elemento sendo tomado de cada linha e de cada coluna. Isto implica na restrição de que dois ou mais elementos não podem ser tomados de uma mesma linha ou coluna. Se os elementos são tomados das linhas em ordem, isto é,

$$a_{1j_1},\ a_{2j_2},\ a_{3j_3},\ \ldots,\ a_{nj_n}$$

então o sinal de cada produto é dado por $(-1)^p$, onde $\underline{p}$ é o nú mero total de inversões na seqüência dos subscritos das colunas $j_1, j_2, j_3, \ldots, j_n$, isto é, o número total de vezes que algum subscrito j_i precede um subscrito menor.

Consideremos a avaliação de um determinante de ordem três,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{matrix} (-1)^0 a_{11}a_{22}a_{33} + (-1)^1 a_{11}a_{23}a_{32} + (-1)^1 a_{12}a_{21}a_{33} + \\ (-1)^2 a_{12}a_{23}a_{31} + (-1)^2 a_{13}a_{21}a_{32} + (-1)^3 a_{13}a_{22}a_{31} \end{matrix}$$

ou $\det A = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}{}^{2}{}_{32} -$

$- a_{13}a_{22}a_{31}$

Analisemos o sexto termo, por exemplo, onde os valores de j são 3, 2, e 1. Existem 3 inversões, por isso temos o sinal negativo para aquele termo.

Após estas definições básicas passemos a analisar as operações com matrizes.

3.2 ADIÇÃO DE MATRIZES

Por definição, os elementos c_{ij} de C=A + B são formados pela regra $c_{ij} = a_{ij} + b_{ij}$. As matrizes A e B devem ter as mesmas dimensões. A soma de matrizes é associativa e comutativa, isto é:

$$(A + B) + D = A +(B + D)$$

$$(A + B) = (B + A)$$

A subtração é dada por $c_{ij} = a_{ij} - b_{ij}$.

Como conseqüência, a matriz resultante da subtração A-A é chamada de matriz zero ou nula. Ela possui as mesmas dimensões de A, porém todos os seus elementos são nulos.

Podemos expressar a soma de duas matrizes A e B guardando o resultado em C de dois modos básicos: por linha e por coluna, chegando aos mesmos resultados finais. Porém o mesmo não acontece quanto à eficiência dos algorítmos quando implementados em FORTRAN conforme análises efetuadas no capítulo anterior. Analisemos os dois casos a seguir:

I) $c_{ij} = a_{ij} + b_{ij}$; $i=1,2,\ldots,m$; $j=1,2,\ldots,n$

II) $c_{ij} = a_{ij} + b_{ij}$; $j=1,2,\ldots,n$; $i=1,2,\ldots,m$

Imaginemos que em ambos os casos tenhamos A, B e C com dimensões de (128, 128) respectivamente, e que só existe espaço na memória principal para uma página de cada matriz considerada nos algorítmos.

<table>
<tr><th>Algorītmo I</th><th>Algorītmo II</th></tr>
<tr><td><pre>
    DO 10 I=1, NLINHA
           DO 20 J=1, NCOLUN
           C(I,J)=A(I,J) + B(I,J)
20   CONTINUE
10 CONTINUE
</pre></td><td><pre>
    DO 10 J=1, NCOLUN
           DO 20 I=1, NLINHA
           C(I,J)=A(I,J) + B(I,J)
20   CONTINUE
10 CONTINUE
</pre></td></tr>
<tr><td>Demanda de pāginas=3np=2048x3=6144
(DP1)</td><td>Demanda de pāginas=3p=16x3=48
(DP2)</td></tr>
</table>

A simples formulação do algorītmo de duas formas diversas, por linha e por coluna conduziu a duas situações bem diversas em termos de sobrecarga para o sistema operacional da māquina hipotētica do exemplo. Ē interessante lembrar, no entanto, que se fosse possīvel, por algum artifīcio manter todas as matrizes integralmente residentes na memōria principal, os dois valores de DP1 e DP2 seriam iguais.

3.3 MULTIPLICAÇÃO ESCALAR

O produto de um escalar c por uma matriz A ē definido como uma matriz cujos elementos são resultantes da multiplicação $c(a_{ij})$. A multiplicação escalar possuī as seguintes propriedades: distribuitiva, associativa e comutativa.

a. $(c + d)A = cA + dA$

b. $c(A + B) = cA + cB$

c. $(cd)A = c(dA)$

d. $A(cB) = (cA)B = c(AB)$

Assim como no caso da adição, podemos implementar o algorītmo de dois modos distintos a saber:

Algorītmo I	Algorītmo II
DO 10 I=1, NLINHA DO 20 J=1, NCOLUN A(I,J)=A(I,J) * C 20 CONTINUE 10 CONTINUE	DO 10 J=1, NCOLUN DO 20 I=1, NLINHA A(I,J)=A(I,J) * C 20 CONTINUE 10 CONTINUE
DP1 = np = 2048	DP2 = p = 16

Podemos observar que este caso de multiplicação escalar ē bem similar ao de adição em termos de demanda de pā ginas ao sistema.

3.4 PRODUTO DE MATRIZES

A definição de produto de matrizes ē mais simples de ser apresentada se analizarmos o seguinte exemplo:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = c_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = c_2$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = c_3$$

Neste sistema de três equações lineares simultâneas, podemos distinguir três matrizes. A matriz de coeficientes A, o vetor X e o vetor C. Podemos então expressar o sistema como AX=C. Outro modo ainda seria:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

A matriz A multiplicada pelo vetor X é igual ao vetor C. Para conseguir o valor de c_i basta multiplicar a linha i da matriz A pelo vetor X, que é um vetor coluna. A multiplicação é realizada termo a termo, e em seguida os produtos individuais são somados, como se pode ver no exemplo. A multiplicação de um vetor por uma matriz pode ser analizada nos dois casos a seguir:

Algorītmo I	Algorītmo II

<pre>   DO 10 I=1, NLINHA
             DO 20 J=1, NCOLUN
             C(I) = C(I)+A(I,J)*X(J)
20   CONTINUE
10   CONTINUE</pre>
	<pre> DO 10 J=1, NCOLUN
	             DO 20 I=1, NLINHA
	             C(I)=C(I)+X(J)*A(I,J)
	20 CONTINUE
	10 CONTINUE</pre>

DP1 = np = 2048	DP2 = p = 16

Nos dois algorītmos apresentados, o vetor C(i) deve ser zerado antes de entrar no primeiro comando DO. Em alguns sistemas, como no B6700, todas as variáveis são zeradas pelo próprio compilador se nenhum valor lhes for dado.

Podemos passar agora ao caso mais geral de produto de matrizes. Em muitos casos a segunda matriz se cons-

titui de mais de uma coluna. O produto matricial pode ser encarado como uma extensão do caso já analisado nesta introdução.

Seja A uma matriz de (m,p) e B uma matriz de (p,n). O produto AB é uma matriz (m,n) denotada por C onde cada elemento c_{ij} de C é obtido multiplicando os elementos correspondentes à i-ésima linha de A pelos elementos da j-ésima coluna de B e, logo após, somando os resultados. Deste modo, cada elemento c_{ij} da matriz C é formado pela regra:

$$c_{11} = a_{11}b_{11} + a_{12}b_{12} + \dots + a_{1p}b_{1p} \quad ,$$

este seria o elemento c_{11} da matriz C. Um elemento qualquer c_{ij} seria assim:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{ip}b_{pj} \quad .$$

De uma forma geral podemos apresentar cada elemento resultante do produto de A por B dando a matriz C do seguinte modo:

$$c_{ij} = \sum_{k=1}^{p} a_{ik}b_{kj} \quad , \text{ para } i=1,2,\dots,m \;;\; j=1,2,\dots,n$$

para A(m,p), B(p,n) e C(m,n).

O produto de A por B não existe senão quando o número de colunas de A é igual ao número de linhas de B. Neste caso, A e B são chamadas matrizes conformes para a multiplicação.

A multiplicação de matrizes é associativa ou

seja:

$$(BC)A = (AB)C$$

para matrizes que sejam conformes.

A multiplicação de matrizes não é comutativa.

As leis de distributividade são atendidas pela multiplicação de matrizes ou seja:

$$C(A+B) = CA + CB$$

$$(A+B)D = AD + BD$$

Em virtude da multiplicação não ser comutativa é necessário especificar a ordem das matrizes sendo multiplicadas. A operação de pré-multiplicação de B por A é o resultado de AB; a pós-multiplicação de B por A é o resultado de BA.

Passemos agora a analisar a multiplicação de matrizes em termos de possíveis algorítmos, como nos casos anteriores. Para análise dos algorítmos uma hipótese foi assumida; existe memória principal suficiente para acomodar uma página para cada matriz, ou seja há espaço para três páginas na memória principal.

Algorítmo I	Algorítmo II
DO 40 I=1, NLINHA DO 40 J=1, NCOLUN C(I,J) = 0.0 DO 30 K=1, NCOLUN 30 C(I,J)=C(I,J)+A(I,K)* B(K,J) 40 CONTINUE	DO 40 J=1, NCOLUN DO 10 I=1, NLINHA 10 C(I,J) = 0,0 DO 30 K=1, NCOLUN DO 20 I=1, NLINHA 20 C(I,J)=C(I,J)+A(I,K)* B(K,J) 30 CONTINUE 40 CONTINUE
DP1 = n*p*(2+p) = 36.864	DP2 = p*(2+p) = 288

Agora que analisamos as principais operações básicas com matrizes, como soma, subtração, multiplicação de matriz por vetor e multiplicação de duas matrizes, devemos ressaltar que levamos em conta matrizes retangulares onde praticamente todos, ou quase todos, os elementos eram diferentes de zero. No caso de matrizes com características diferentes, cuidados especiais devem ser tomados com relação a armazenamento e manipulação, tendo-se em vista eficiência de processamento e minimização de paginação. Passemos então à seção seguinte onde vamos analisar tipos especiais de matrizes, suas características e impacto sobre os algorítmos mais comuns.

3.5 TIPOS ESPECIAIS DE MATRIZES

Em álgebra linear, por exemplo, existem muitas aplicações para certos tipos de matrizes.

Como já foi visto antes, uma matriz que possua o mesmo número de linhas e de colunas é chamada de matriz quadrada. Em uma matriz quadrada os elementos A(1,1), A(2,2), ..., A(n,n) formam a diagonal principal da matriz. Uma matriz diagonal é uma matriz que contém zeros em todos os elementos, exceto aqueles que pertencem à diagonal principal. Em uma matriz diagonal onde todos os elementos possuem o mesmo valor é chamada de matriz escalar. Um caso especial de matrizes escalares é aquele em que este número é 1. Ela é chamada de matriz identidade ou matriz unitária.

Uma matriz com todos os elementos iguais a zero é chamada de matriz nula.

As matrizes triangulares correspondem a outro caso importante das matrizes quadradas, onde todos os elementos de um lado da diagonal principal são nulos e do outro são diferentes de zero. Quando em uma matriz A, os elementos $a_{ij}=0$ sempre que i for maior que j teremos uma matriz triangular superior. No caso em que $a_{ij}=0$ para i menor que j teremos matrizes triangulares inferiores.

As matrizes idempotentes são aquelas que obedecem à regra que diz que $A^n=A$, para n inteiro e maior ou igual a 1.

Matrizes nulipotentes são aquelas onde $A^p=0$, porém $A^{p-1}\neq 0$. Neste caso a matriz A é chamada de nulipotente de

grau p.

Outra classe de matrizes é a das matrizes transpostas, onde sendo dada uma matriz A, de n linhas e m colunas, podemos obter a partir de A uma nova matriz de dimensões m por n, chamada transposta de A, trocando-se as linhas pelas colunas respectivamente, de tal modo que para uma matriz $A=(a_{ij})$ tenhamos a nova matriz $A=(a_{ji})$. Isto pode ser visto como uma operação de reescrever os vetores-linha em vetores-coluna.

A matriz transposta será representada neste trabalho por A^T.

Uma matriz quadrada é chamada de simétrica quando $A=A^T$. Esta denominação vem do fato de que os elementos de A serem simétricos em relação à diagonal principal.

Quando acontece o fato de uma matriz quadrada A ser igual a $-A^T$ chamamos a matriz A de anti-simétrica. Nas matrizes anti-simétricas temos $a_{ij}=-a_{ji}$ se $i \neq j$ e que $a_{ij}=0$ se $i=j$.

Um detalhe interessante é que qualquer matriz quadrada pode ser decomposta na soma de uma matriz simétrica e de uma matriz anti-simétrica e tal decomposição é única. A matriz simétrica S pode ser calculada por:

$$S = (A + A^T) / 2 \quad ,$$

e a matriz anti-simétrica P pode ser calculada por:

$$P = (A - A^T) / 2 \quad .$$

Passemos a analisar o caso de uma matriz A(m,n)

e de outra matriz B(m,m) tal que AB = BA = I_m. Neste caso a matriz B é chamada de inversa da matriz A. A matriz I_m é a matriz identidade ou unitária.

Vamos representar a matriz inversa neste trabalho por A^{-1}.

Se uma matriz não admite inversa ela é chamada de matriz singular, ou não-inversível. Se em uma matriz A(n,n) existe uma matriz coluna X não nula, onde AX=0, então a matriz A não admite inversa.

Após todas estas definições passemos a analisar como alguns destes tipos especiais de matrizes podem ser armazenados visando otimizar o uso de memória. De um modo geral, as matrizes comumente usadas possuem todos os seus elementos armazenados. Para as matrizes simétricas somente uma parte da matriz necessita ser armazenada, a parte superior ou a inferior, em posições seqüenciais de memória, permitindo deste modo uma substancial redução do uso de memória. Para as matrizes diagonais somente os elementos da diagonal são armazenados, visto que todos os demais são nulos.

Passemos a analisar como estes tipos especiais de matrizes podem ser armazenados sequencialmente na memória.

As matrizes simétricas e as matrizes triangulares superiores ou inferiores podem ser armazenadas do mesmo modo. Somente a parte que seja diferente de zero precisa ser guardada nas matrizes triangulares. Uma matriz simétrica de ordem n pode ser armazenada em um vetor de comprimento N*(N+1)/2 ao invés de N*N posições como em outros tipos de matrizes. Para uma matriz de ordem 128, usaríamos apenas 8256

posições ao invés de 16384 para armazenar toda a matriz. Isto representa um ganho de cerca de 50 por cento em matrizes de maior tamanho. Veja a figura 3.1.

Quando passamos a usar esta técnica em matrizes diagonais os ganhos de memória são ainda maiores. Matrizes diagonais de ordem n podem ser armazenadas em apenas n posições.

Este é o processo utilizado nas subrotinas ARRAY e LOC pertencentes ao IBM Scientific Subroutine Package para armazenamento e manipulação de matrizes de modo a otimizar o uso de memória principal.

Os algorítmos apresentados para realizar as operações com matrizes podem ser adaptados sem nenhuma dificuldade pelos leitores, por isso não os apresentaremos aqui, apesar de os termos testados também, com um substancial ganho de memória e minimização das operações de paginação usando os mesmos conceitos apresentados no capítulo dois.

Em todos os tipos de matrizes citados, procurou-se ressaltar suas principais características sem no entanto constituir um estudo exaustivo. Maiores informações podem ser obtidas em |15|, |18|, |19| e em |21|.

Deixamos propositadamente o tratamento de matrizes esparsas para o capítulo cinco deste trabalho.

3.6 TRATAMENTO DE MATRIZES POR PARTIÇÃO EM BLOCOS

Quando os coeficientes de uma matriz qualquer não podem ser armazenados de uma só vez na memória principal,

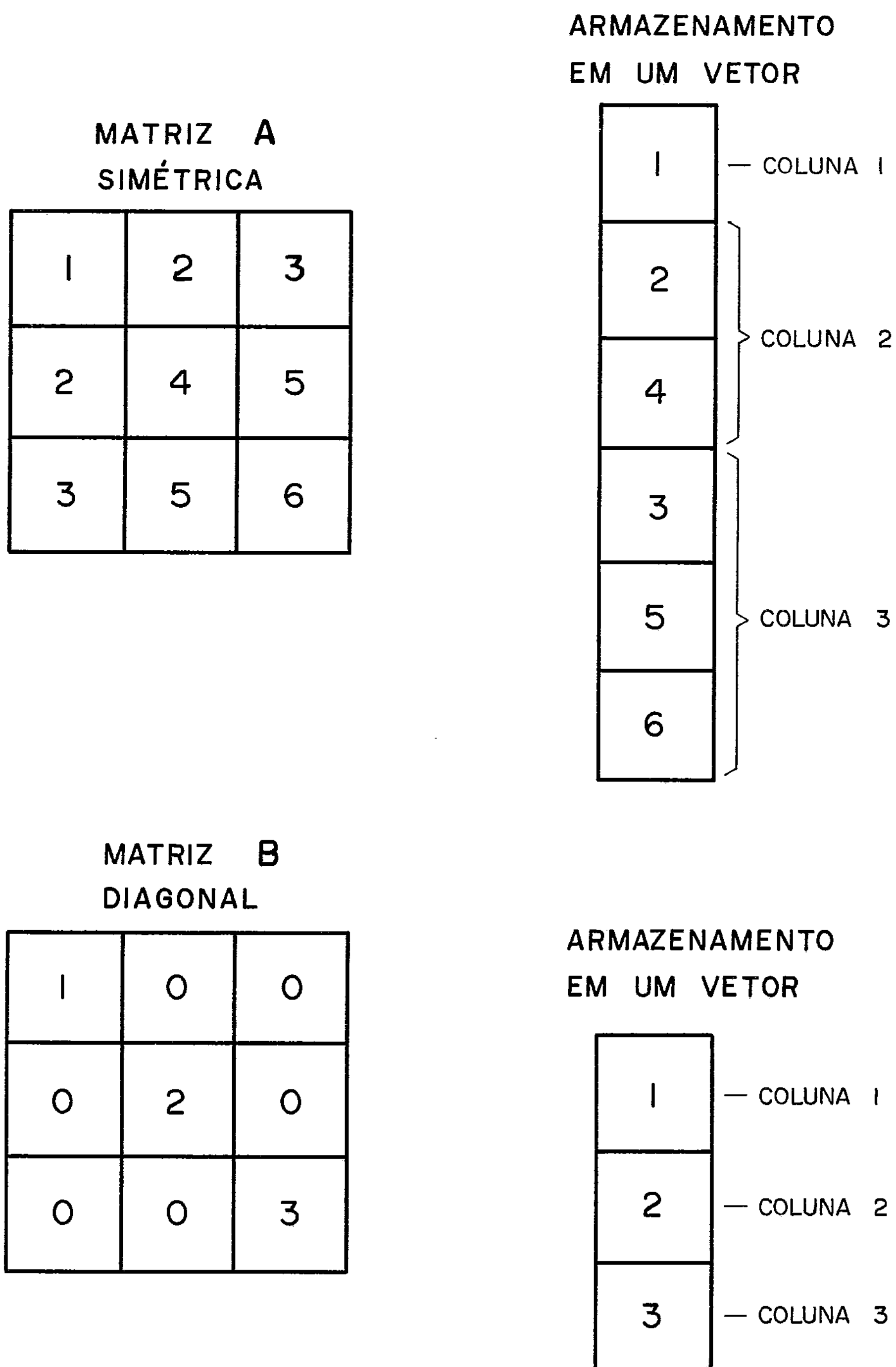


Figura 3.1 – Armazenamento de matrizes especiais em vetor (Retirado de [15])

por limitação de sua configuração, o que é comum em pequenas máquinas, podemos utilizar a técnica de dividir esta matriz em blocos de dimensões que possam ser tratados na memória principal disponível. Apenas os blocos necessários ao processamento em um dado instante serão mantidos residentes e todos os demais serão armazenados em memória auxiliar.

Existem diversas proposições na literatura de como efetuar esta partição e a eficiência de cada esquema é tanto maior quanto menor for o gasto de memória principal e o de transferências para a memória auxiliar e vice-versa. Algumas referências mais profundas podem ser encontradas em |18| e em |22|.

A filosofia básica é bem simples e para uma qualquer matriz A, podemos subdividí-la em matrizes menores chamadas blocos ou submatrizes de A. Dada uma matriz qualquer, ela pode ser subdividida em blocos de diversos modos diferentes. Vejamos em seguida um exemplo de como tal operação pode ser realizada:

Exemplo: Seja a matriz A

$$A = \left[\begin{array}{cccc|cc} 3 & 6 & 2 & 1 & 9 & 8 \\ 1 & 8 & 7 & 6 & 8 & 3 \\ 5 & 4 & 3 & 4 & 7 & 5 \\ \hline 0 & 1 & 4 & 3 & 5 & 2 \\ 1 & 5 & 2 & 7 & 6 & 1 \end{array}\right]$$

Se definirmos,

$$N_{11} = \begin{bmatrix} 3 & 6 & 2 & 1 \\ 1 & 8 & 7 & 6 \\ 5 & 4 & 3 & 4 \end{bmatrix} , \quad N_{12} = \begin{bmatrix} 9 & 8 \\ 8 & 3 \\ 7 & 5 \end{bmatrix}$$

$$N_{21} = \begin{bmatrix} 0 & 1 & 4 & 3 \\ 1 & 5 & 2 & 7 \end{bmatrix} , \quad N_{22} = \begin{bmatrix} 5 & 2 \\ 6 & 1 \end{bmatrix}$$

a matriz A pode ser escrita na seguinte forma:

$$\begin{bmatrix} N_{11} & N_{12} \\ N_{21} & N_{22} \end{bmatrix}$$

onde N_{11}, N_{12}, N_{21} e N_{22} são submatrizes da matriz original A. Observemos que N_{11} e N_{21} possuem o mesmo número de colunas, e N_{12} e N_{22} igualmente o mesmo número de colunas. Podemos verificar também que N_{11} e N_{12}, e N_{21} e N_{22} também possuem o mesmo número de linhas.

Consideremos agora o caso de uma matriz M, de r linhas e de c colunas, a qual desejamos subdividir em 4 blocos da seguinte maneira:

$$M_{r,c} = \begin{bmatrix} A_{p,q} & B_{p,(c-g)} \\ C_{(r-c),q} & D_{(r-p),\ (c-q)} \end{bmatrix}$$

onde A, B, C, e D são submatrizes de ordem pxq, px(c-g),

(r-p)xq, e (r-p)x(x-q) respectivamente.

No caso mais geral de uma matriz Z, de dimensões pxq, ela pode ser subdividida em r linhas e c colunas de submatrizes como:

$$M = \begin{bmatrix} M_{11} & M_{12} & M_{13} & \cdots & M_{1c} \\ M_{21} & M_{22} & M_{23} & \cdots & M_{2c} \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ M_{r1} & M_{r2} & M_{r3} & \cdots & M_{rc} \end{bmatrix}$$

onde M_{ij} é a submatriz na i-ésima linha e j-ésima coluna de submatrizes. Se a i-ésima linha de submatrizes tem p_i linhas de elementos e a j-ésima coluna de submatrizes tem q_j colunas, então M_{ij} é de ordem $p_i \times q_j$, onde temos as seguintes relações:

$$p_1 + p_2 + \ldots + p_r = p$$

e também,

$$q_1 + q_2 + \ldots + q_r = q.$$

Os resultados das operações sobre matrizes divididas em blocos, podem ser obtidos fazendo-se as operações sobre os blocos, como se fossem os elementos da matriz.

Dada uma matriz M subdividida em blocos como a seguir:

$$M = \begin{bmatrix} M_{11} & M_{12} & \cdots\cdots\cdots & M_{1n} \\ M_{21} & M_{22} & \cdots\cdots\cdots & M_{2n} \\ . & . & \cdots\cdots\cdots & . \\ M_{m1} & M_{m2} & \cdots\cdots\cdots & M_{mn} \end{bmatrix}$$

se multiplicarmos cada bloco por um escalar $\underline{k}$, todos os elementos de M ficam multiplicados por $\underline{k}$, como mostrado a seguir:

$$kM = \begin{bmatrix} kM_{11} & kM_{12} & \cdots\cdots\cdots & kM_{1n} \\ kM_{21} & kM_{22} & \cdots\cdots\cdots & kM_{2n} \\ . & . & \cdots\cdots\cdots & . \\ kM_{m1} & kM_{m2} & \cdots\cdots\cdots & kM_{mn} \end{bmatrix}$$

Suponhamos agora, que é dada uma outra matriz $\underline{N}$ subdividida no mesmo número de blocos que a matriz $\underline{M}$ anterior, e que os blocos de M e de N possuem os mesmos tamanhos, ou as mesmas dimensões. Somando os blocos correspondentes, estaremos somando os elementos correspondentes de M e de N, e a matriz resultante será da seguinte forma:

$$M+N = \begin{bmatrix} M_{11}+N_{11} & M_{12}+N_{12} & \cdots\cdots\cdots & M_{1n}+N_{1n} \\ M_{21}+N_{21} & M_{22}+N_{22} & \cdots\cdots\cdots & M_{2n}+N_{2n} \\ . & . & \cdots\cdots\cdots & . \\ M_{m1}+N_{m1} & M_{m2}+N_{m2} & \cdots\cdots\cdots & M_{mn}+N_{mn} \end{bmatrix}$$

3.6.1 MULTIPLICAÇÃO DE MATRIZES SUBDIVIDIDAS EM BLOCOS

A subdivisão de matrizes em blocos é de grande utilidade na multiplicação de matrizes, pois se duas matrizes A e B, são divididas em submatrizes, as quais são apropriadamente conformáveis para a multiplicação, o produto AB pode ser expresso como uma matriz subdividida, tendo submatrizes que são funções das submatrizes de A e de B. Existe um teorema que diz que é possível multiplicar matrizes "repartidas" como se as submatrizes fossem elementos (escalares) sempre que a partição seja feita de tal forma que os produtos apropriados possam ser feitos. Em alguns casos simples isto pode ser visto sem dificuldade como:

a) $$\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} B = \begin{bmatrix} A_1 B \\ A_2 B \end{bmatrix}$$

b) $$A \begin{bmatrix} B_1 & B_2 \end{bmatrix} = \begin{bmatrix} AB_1 & AB_2 \end{bmatrix}$$

c) $$\begin{bmatrix} A_1 & A_2 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} A_1B_1 + A_2B_2 \end{bmatrix}$$

O caso geral segue por repetições sucessivas os três casos apresentados. Os detalhes são bastante laboriosos e sugerimos ao leitor que verifique os casos gerais em |18|, em |22|, em |9| ou em |11|.

Analisemos um exemplo de como tais operações podem ser realizadas para as matrizes A e B do caso a seguir:

$$A = \left[\begin{array}{cc|c} 3 & 1 & 0 \\ -2 & 5 & 4 \\ \hline 3 & 7 & 17 \end{array}\right] \qquad B = \left[\begin{array}{cc|c} 4 & 6 & 0 \\ 3 & 1 & -1 \\ \hline 2 & 0 & 1 \end{array}\right]$$

e também temos as submatrizes seguintes:

$$A_{11} = \begin{bmatrix} 3 & 1 \\ -2 & 5 \end{bmatrix}, \; A_{12} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}, \; A_{21} = \begin{bmatrix} 3 & 7 \end{bmatrix}, \; A_{22} = \begin{bmatrix} 17 \end{bmatrix},$$

$$B_{11} = \begin{bmatrix} 4 & 6 \\ 3 & 1 \end{bmatrix}, \; B_{12} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \; B_{21} = \begin{bmatrix} 2 & 0 \end{bmatrix}, \; B_{22} = \begin{bmatrix} 1 \end{bmatrix}$$

As matrizes A e B podem ser expressas então do seguinte modo:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad \text{e} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Caso A_{ij} e B_{ij} para $i=1,2$ e $j=1,2$ fossem escalares, teríamos que:

$$\begin{bmatrix} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \end{bmatrix}$$

A partir do produto das matrizes originais A e B ou então a partir das multiplicações das submatrizes como mostrado acima chegaríamos ao mesmo resultado que é o seguinte:

$$AB = \begin{bmatrix} 15 & 19 & -1 \\ 15 & -7 & -1 \\ 67 & 25 & 10 \end{bmatrix}$$

como podem os leitores verificar com facilidade.

Esta facilidade de subdividir matrizes em blocos ou submatrizes é muito útil quando por exemplo em sistemas físicos que podem ser subdivididos em subsistemas com interconexões, o comportamento do sistema todo pode ser freqüentemente descrito por uma matriz global repartida de tal maneira que as submatrizes ao longo da diagonal descrevem as partes separadas de todo o sistema, e as submatrizes que não estão na diagonal representam as interconexões dos subsistemas. Isto pode ser usado para estudar a estrutura de um sistema complexo.

É útil dividir uma matriz em submatrizes quando alguns dos blocos forem a matriz 0 (zero). Vejamos o exemplo a seguir:

$$M = \left[\begin{array}{cc|ccc} -1 & 4 & 17 & -2 & 0 \\ 2 & 0 & 5 & 3 & 1 \\ \hline 0 & 0 & 2 & 4 & 3 \\ 0 & 0 & 0 & -1 & 5 \end{array}\right]$$

e temos então:

$$M = \begin{bmatrix} M_{11} & 0 \\ 0 & M_{22} \end{bmatrix}$$

e podemos dizer que M é a soma direta de M_{11} e de M_{22}.

Outra vantagem da partição de matrizes em blocos é o fato de poder-se estudar as suas propriedades, examinando as propriedades das submatrizes.

Para o escopo deste trabalho estas noções básicas sobre divisão de matrizes por blocos são suficientes e não desceremos a um nível de detalhamento mais profundo, o que deixamos ao cargo do leitor que poderá se utilizar das referências já fornecidas anteriormente.

Passemos agora ao estudo dos sistemas de equações lineares e dos métodos utilizados para sua solução.

4. SISTEMAS DE EQUAÇÕES LINEARES

A solução de um sistema de equações lineares simultâneas pode ser necessária para se resolver problemas variados em engenharia como por exemplo: análise de tensões em uma estrutura, o estudo de circuitos elétricos complexos, a solução de problemas de misturas químicas, e análise de vibrações em sistemas mecânicos.

Geralmente tais sistemas são dados na forma AX=C onde os a_{ij} são coeficientes conhecidos, os c_i são constantes conhecidas e os x_i são as incógnitas das equações a serem resolvidas.

Neste trabalho trataremos de dois tipos de métodos para solucionar um sistema de equações lineares, os métodos de eliminação e os iterativos. Os métodos de eliminação teoricamente fornecem a solução exata, a menos dos erros de arredondamento. Os métodos iterativos, podem ser de extrema utilidade em muitos problemas práticos onde se busca uma solução, que apesar de não ser exata, se aproxima da solução desejada com um grau de erro aceitável. Em diversos casos para chegar a uma precisão desejada, é necessário um grande número de interações e felizmente para isto temos os computadores à nossa disposição.

Análises mais aprofundadas sobre os tópicos apresentados de forma sintética neste capítulo podem ser encontradas em |16|, |17|, |18|, |19|, |20| e |21|. Procuramos apresentar apenas aqueles conceitos que nos parecem mais importantes para o trabalho de pesquisa em discussão.

4.1 NOÇÕES PRELIMINARES

Os conceitos básicos sobre matrizes apresentados no capítulo anterior serão de grande utilidade em todo o decorrer deste capítulo. Para um sistema AX=B, os coeficientes a_{ij} formam a matriz quadrada A. Os elementos a_{ij} onde i=j, formam a diagonal principal. Uma matriz formada por tais elementos e onde todos os demais sejam nulos forma uma matriz diagonal que representaremos por D. Analogamente, podemos definir duas outras matrizes estritamente triangulares inferior e superior. Chamemos de L a matriz triangular inferior, onde $a_{ij}=0$ se $i=j$ e se $i<j$. Para a matriz onde $a_{ij}=0$ se $i\geqslant j$ usemos o símbolo U que representa a matriz triangular superior. As matrizes D, L e U formam ou correspondem à partições da matriz A, tal que $A = D + L + U$.

Podemos começar agora a falar dos sistemas lineares propriamente ditos.

A solução de um sistema linear é um conjunto de valores das <u>n</u> incógnitas que satisfaz a todas as equações. Se este conjunto é único, o sistema é dito determinando ou não singular. Se existe uma infinidade de tais conjuntos, o sistema é dito indeterminado, e se não existe nenhum conjunto, ele é dito impossível e não tem solução. Para os dois últimos casos citados, o determinante da matriz do sistema é nulo. O sistema AX=B é dito singular se det A=0.

As considerações sobre unicidade ou não da solução são um tanto teóricas, pois ao tratarmos problemas numéricos, podem ocorrer aproximações que nos conduzam a soluções

diversas, e em muitos casos, completamente erradas |16|. Isto pode ocorrer nos casos de quase singularidade, onde o det A é quase nulo. Costuma-se dizer que a matriz de tais sistemas é "mal condicionada". Em outras palavras, o determinante de uma matriz mal condicionada é próximo de zero.

Pode ocorrer em um dado exemplo que tenhamos que resolver um sistema cuja matriz de coefientes seja A e outro cuja matriz de coeficiente seja B. Podemos dizer que A é pior condicionada que B se observarmos que:

$$\frac{|\det A|}{\prod_{i=1}^{n} \left(\sum_{j=1}^{n} |a_{ij}| \right)} < \frac{|\det B|}{\prod_{i=1}^{n} \left(\sum_{j=1}^{n} |b_{ij}| \right)}$$

Mesmo se isto se observar, só poderemos suspeitar que ao resolvermos o sistema A, o efeito de erro será maior do que o respectivo de um sistema B.

Como principal alerta para os problemas que vamos enfrentar, ressaltamos a necessidade de se trabalhar com valores tão exatos quanto possível para os coeficientes do sistema. Porém não podemos nos esquecer que os citados coeficientes são obtidos em muitos casos de modo experimental e logo são aproximados. Outro problema é o efeito de arredondamento na solução. Cuidados devem ser tomados, pois isto pode diminuir a exatidão da solução encontrada.

Para o caso de um sistema AX=C apresentar todos os c_i iguais a zero, as equações são ditas homogêneas.

Os métodos apresentados a seguir são apropria-

dos para sistemas lineares de equações simultâneas, não homogêneas, não singulares e independentes.

4.2 MÉTODOS DE ELIMINAÇÃO

Os métodos de eliminação ou diretos se caracterizam pela anulação de certos coeficientes a_{ij} por meio de adições das equações dos sistemas, modificando seu aspecto primitivo, mas transformando-o em um sistema de solução mais fácil.

Analisemos um exemplo para melhor compreensão dos tópicos seguintes. A solução de um sistema de duas equações simultâneas da forma:

$$a_{11}x_1 + a_{12}x_2 = c_1$$

$$a_{21}x_1 + a_{22}x_2 = c_2$$

poderia ser feita assim:

- Resolva a primeira equação para x_1 em termos de x_2,
- Substitua esta na segunda equação,
- Calcule o valor de x_2 da segunda equação,
- Substitua aquele valor de volta na primeira equação,
- Finalmente calcule o valor de x_1 da primeira equação.

Isto pode ser mostrado assim:

$$x_1 = \frac{1}{a_{11}} (c_1 - a_{12}x_2)$$

$$\frac{a_{21}}{a_{11}} (c_1 - a_{12}x_2) + a_{22}x_2 = c_2$$

$$x_2 = \frac{c_2 - a_{21}c_1/a_{11}}{a_{22} - a_{21}a_{12}/a_{11}} = \frac{a_{11}c_2 - a_{21}c_1}{a_{11}a_{22} - a_{21}a_{12}}$$

$$x_1 = \frac{c_1 - a_{12}((a_{11}c_2 - a_{21}c_1) \; / \; (a_{11}a_{22} - a_{21}a_{12}))}{a_{11}} = \frac{a_{22}c_1 - a_{12}c_2}{a_{11}a_{22} - a_{21}a_{12}}$$

Considerações sobre os valores dos numeradores e denominadores são apresentados posteriormente.

Outro método de solução é o seguinte, onde obviamente chegamos aos mesmos resultados passando por passos diferentes:

- Multiplique a primeira equação por a_{21},
- Multiplique a segunda equação por a_{11},
- Subtraia a segunda equação da primeira e chegamos ao mesmo resultado anterior, como se pode ver.

$$\begin{array}{r} a_{21}a_{11}x_1 + a_{21}a_{12}x_2 = a_{21}c_1 \\ -a_{11}a_{21}x_1 - a_{11}a_{22}x_2 = -a_{11}c_2 \\ \hline (a_{21}a_{12} - a_{11}a_{22})x_2 = a_{21}c_1 - a_{11}c_2 \end{array}$$

$$x_2 = \frac{a_{21}c_1 - a_{11}c_2}{a_{21}a_{12} - a_{11}a_{22}} = \frac{a_{11}c_2 - a_{21}c_1}{a_{11}a_{22} - a_{21}a_{12}}$$

$$x_1 = \frac{a_{22}c_1 - a_{12}c_2}{a_{11}a_{22} - a_{21}a_{12}}$$

Para o caso de c_1 e $c_2$serem iguais a zero, as equações são chamadas homogêneas. Em ambas as equações os denominadores são iguais. Se o denominador for diferente de zero teremos então os valores de x_1 e de x_2. Caso o denominador seja igual a zero, as equações são chamadas de singulares. No caso dos numerados serem iguais a zero, não existe solução para o sistema de equações lineares. Temos ainda o caso em que ambos, o numerador e o denominador são iguais a zero, situação onde existem infinitas soluções para o sistema e as equações são chamadas dependentes, ou seja uma equação pode ser obtida de outra através de algumas operações algébricas.

Os resultados apresentados no exemplo podem ser generalizados para um número qualquer de equações. Um sistema de n equações pode ser tornado homogêneo se os valores de c_i forem tornados iguais a zero. A solução de sistema de equações lineares é um conjunto de valores de x_i que têm a forma seguinte:

$$x_i = f_i(a_{ij}, c_i) \;/\; g(a_{ij}).$$

A função $g(a_{ij})$ é a mesma para cada x_i, e se segue que no caso de tal denominador ser igual a zero, o sistema é dito singular. Finalmente, se o numerador e o denominador são nulos,

as equações não são todas independentes.

4.2.1 MÉTODO DE "GAUSS"

O método usado no exemplo anterior pode ser extendido para um sistema de mais de duas equações. Na prática, no entanto, o método pode ser simplificado e condensado, utilizando-se uma técnica atribuída a GAUSS, que vamos agora analisar.

O método de GAUSS consiste em transformar a matriz do sistema em uma matriz triangular superior. Antes de detalharmos o método citado, vejamos quais operações básicas podem ser executadas em um sistema sem que a solução seja alterada |19|. Elas são as seguintes:

1. A posição de quaisquer das duas equações do sistema pode ser trocada. A ordem na qual as equações são escritas é arbitrária. Porém a ordem na qual elas são resolvidas pode afetar o resultado por causa de erros de arredondamento e truncamento, acumulados durante os cálculos efetuados nos diversos passos.
2. Qualquer equação pode ser multiplicada ou dividida em qualquer instante por uma constante diferente de zero.
3. Qualquer equação pode ser adicionada a outra, e a equação resultante pode então substituir qualquer uma das duas equações originais do sistema. Igualmente as regras 2 e 3 podem ser combinadas, de tal modo que uma equação seja multiplicada por uma constante e em seguida adicionada a qualquer outra equação escolhida. Em virtude de estarmos tra-

balhando com os coeficientes de A e de C, podemos omitir os símbolos x_i das equações, assim como os sinais de igual. A matriz A é chamada de matriz de coeficientes, e a matriz C de vetor de constantes. Para simplificar as expressões, podemos chamar c_1 de $a_{1, n+1}$, c_2 de $a_{2, n+2}$, e assim sucessivamente até c_n de $a_{n, n+1}$. A matriz dos elementos a_{ij} agora será chamada de matriz extendida A(n, n+1), onde a última coluna é o vetor C. No método de eliminação de GAUSS, em cada passo a matriz é transformada por algumas operações nas linhas. A linha usada para alterar as outras linhas é chamada de linha pivô. O elemento usado em cada passo, a_{ij}, é chamado de elemento pivô.

O procedimento usado no símbolo de eliminação de GAUSS é o seguinte: começamos as transformações anulando os elementos abaixo de a_{11} da matriz, ou seja, os elementos da mesma coluna. Isto é chamado de 1a. eliminação e o elemento a_{11} é chamado de pivô dessa eliminação, que é realizada do seguinte modo 16 :

- Dividir a linha 1 (ℓ_1) pelo pivô, o que gera uma nova ℓ_1.
- Somar ℓ_2 a ℓ_1 pré-multiplicada por $-a_{21}$, de modo a eliminar o elemento a_{21} da matriz.

O processo pode ser extendido a todas as linhas de modo que ao final da primeira eliminação a matriz extendida estará transformada de tal modo que todos os elementos da primeira coluna, abaixo de a_{11} estarão transformados em zero.

A segunda eliminação consiste em eliminar todos

os elementos da segunda coluna abaixo de a_{22} denominado pivô desta eliminação. Para realizá-la basta repetir o procedimento da primeira eliminição. Ao final da segunda eliminação teremos todos os elementos da segunda coluna, abaixo de a_{22} tornados nulos. Prosseguimos até alcançar a última linha repetindo em cada passo os mesmos procedimentos vistos. Ao final teremos uma matriz triangular superior e podemos, sem dificuldades, resolver as equações do sistema. Em resumo, ao final da última eliminação a matriz estava triangularizada. Visto de outro modo, a fim de cada eliminação um novo elemento era tornado igual a 1 na diagonal principal e todos os elementos abaixo dele tornados iguais a zero na mesma coluna. As operações de eliminação visam a triangularização da matriz de coeficientes. Em seguida realizamos a substituição dos valores calculados de x_i a partir da última equação até a primeira, ou seja de baixo para cima. Esta substituição ao contrário ou de baixo para cima, é referenciada na literatura como "Backward substitution" e a eliminação como "Forward elimination". Ao invés de se realizar a substituição ao contrário para determinar os valores de x_i, existe um procedimento alternativo de continuar usando o processo de eliminação para converter os coeficientes da matriz de tal forma que os elementos da diagonal sejam diferentes de zero e os demais todos nulos, exceto aqueles que fazem parte do vetor de constantes, onde estarão as soluções.

É importante observar que nenhum dos elementos que servem de pivô numa eliminação pode ser igual a zero. Se isto ocorrer, devemos trocar a posição da linha em questão

com as que estão abaixo, de tal modo que tais elementos não sejam nulos. Esta operação não altera a solução do sistema. Só não podemos trocar a linha em questão por linhas que já tenham sido tratadas e que estejam acima, pois isto afetaria o resultado das eliminações anteriores. Quando este procedimento não for possível porque todos os elementos da coluna do pivô abaixo da diagonal principal são nulos, o sistema é singular.

O método de GAUSS de eliminação, como mencionado antes para diagonalizar a matriz de coeficientes pode ser visto no fluxograma apresentado na figura 4.1.

No fluxograma, k é o número da linha sendo usada para eliminar os coeficientes da k-ésima coluna nas linhas mais baixas. Antes de ser usada, no entanto, a linha é dividida pelo seu próprio elemento da diagonal a_{kk} de tal modo que a matriz final será formada por elementos iguais a 1 na diagonal principal e a coluna das constantes dará diretamente os valores das incógnitas. Esta divisão é mostrada na primeira coluna. A eliminação é dada na segunda coluna do fluxograma através de três comandos DO embutidos. A eliminação ao contrário, está nos três comandos DO embutidos da terceira coluna do fluxograma. Os índices i e j referenciam a linha e a coluna do elemento a_{ij}. O cálculo do fator s é realizado fora de cada ciclo. Caso este passo fosse incluído no ciclo, os elementos da diagonal seriam alterados antes que fossem usados para modificar os elementos da linha em questão.

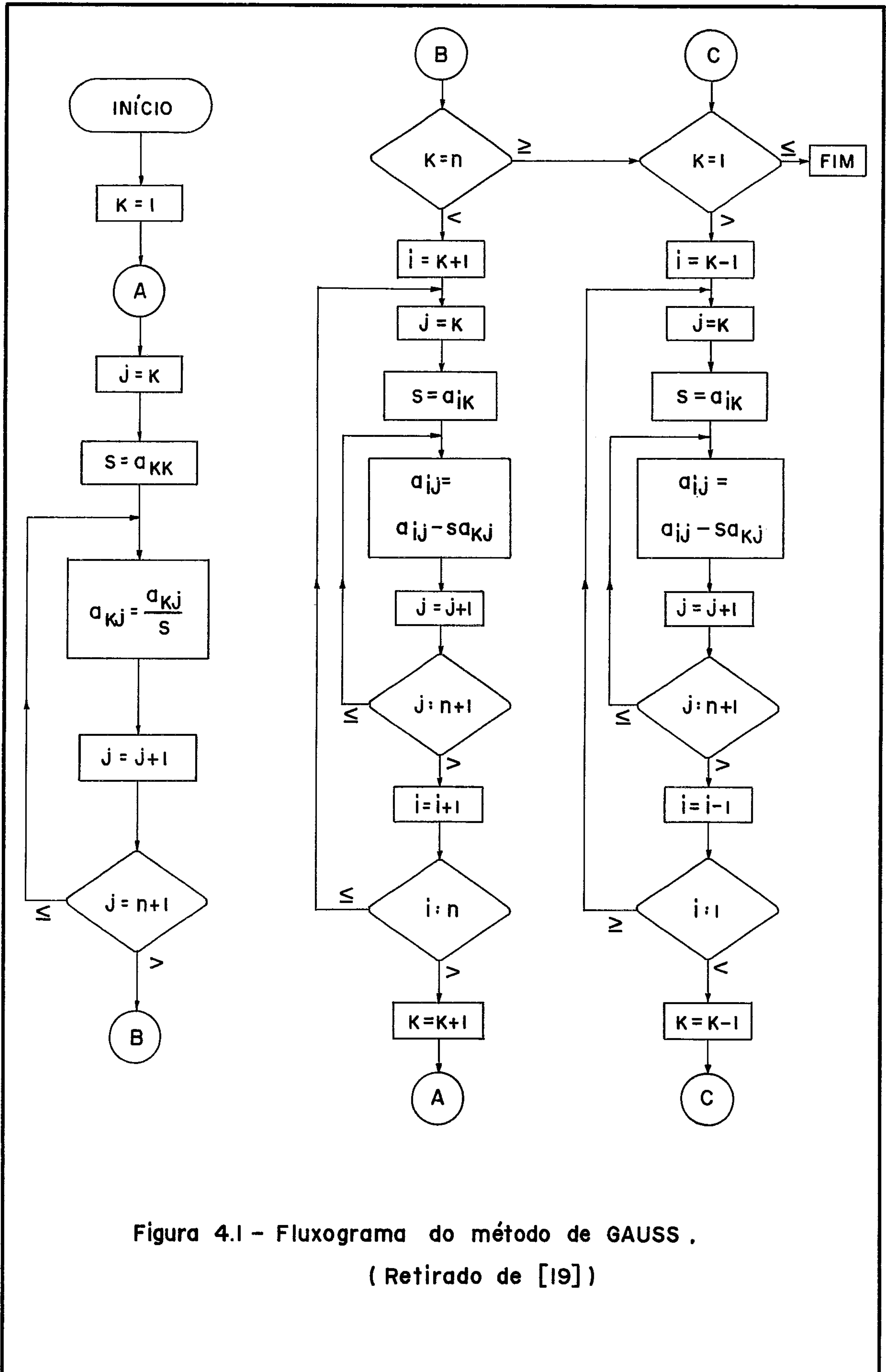


Figura 4.1 – Fluxograma do método de GAUSS.
(Retirado de [19])

4.2.2 MÉTODO DE ELIMINAÇÃO DE GAUSS-JORDAN

As eliminações vistas no método anterior podem ser combinadas de tal modo que formem um procedimento único conhecido como o método de GAUSS-JORDAN. A variação em relação ao método anterior é que a k-ésima linha é usada para eliminar os elementos da k-ésima coluna, não somente abaixo do elemento a_{kk}, mas também acima dele. Assim, ao fim de um único passo ao longo da matriz, os únicos elementos que sobram são os elementos do vetor coluna e os elementos da diagonal valendo 1.

O método de GAUSS-JORDAN é mostrado no fluxograma da figura 4.2 onde a notação é a mesma utilizada no fluxograma da figura 4.1. O ciclo de eliminação principal sempre começa na linha 1 e prossegue até a linha n, pulando a linha k, que é a que está sendo usada para a linha pivô.

Alguns dos passos indicados nas figuras 4.1 e 4.2 não são realmente necessários para a obtenção da solução. Uma análise detalhada permite verificar que o armazenamento dos zeros na matriz de coeficientes é trabalho desnecessário, porque tais elementos nunca mais serão consultados ou usados. Os únicos elementos que interessam ao final da operação mostrada, são os elementos da última coluna, o vetor das constantes da matriz extendida, que foi transformado pelas operações com as linhas em um vetor que contém as soluções. Um novo fluxograma é apresentado na figura 4.3 com estas modificações no método de GAUSS-JORDAN.

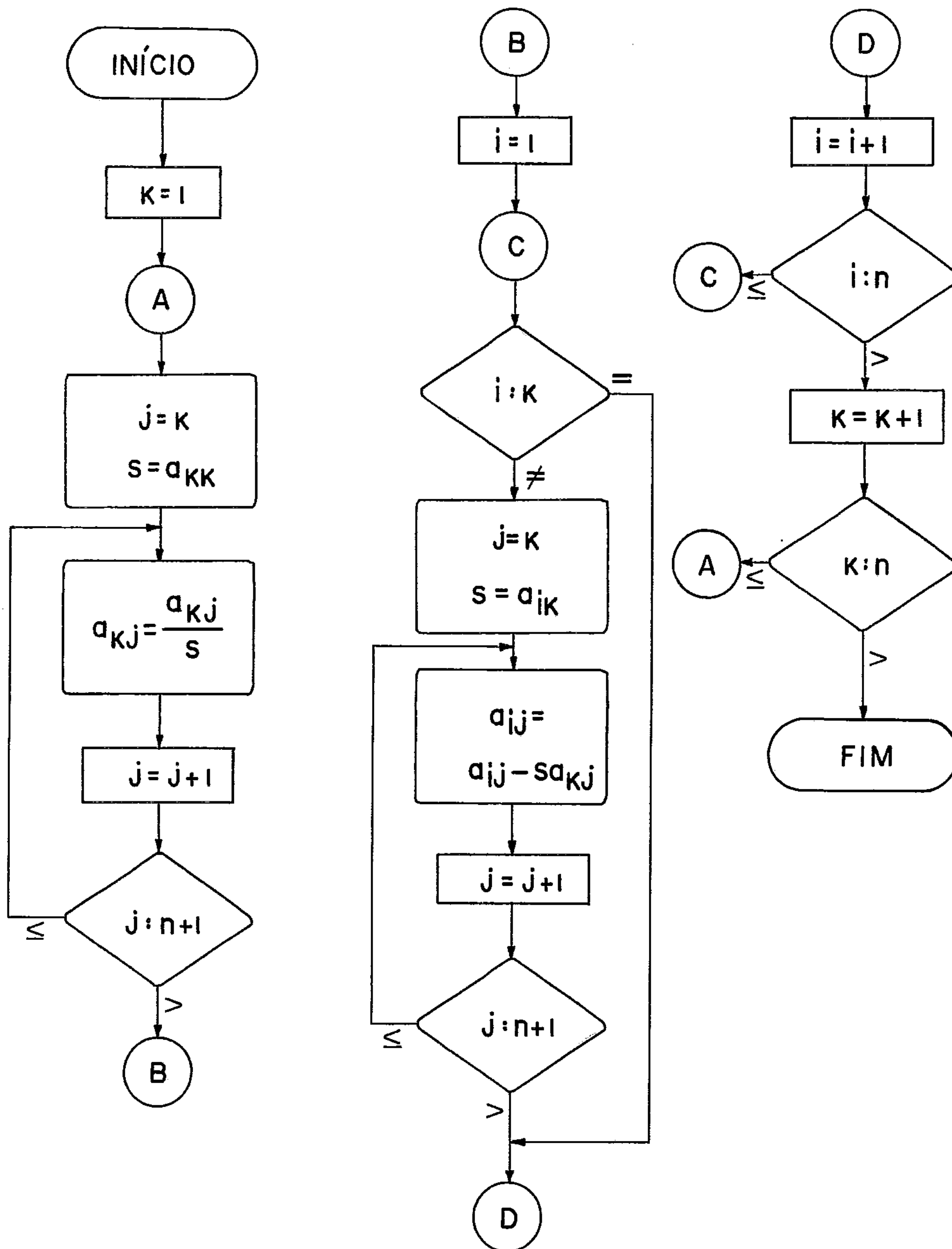


Figura 4.2 – Fluxograma do método de GAUSS-JORDAN

(Retirado de [19])

4.2.3 PROBLEMAS DE ARREDONDAMENTO

A solução de um sistema de equações lineares pode ser muito prejudicada pelos erros que geralmente ocorrem causados por truncamentos e arredondamentos durante o processamento, devido ao fato de que os computadores utilizam um número fixo de dígitos para armazenar cada número. Um caso que acontece com alguma freqüência é aquele em que o elemento pivô não é zero, mas é muito pequeno. Os erros cometidos durante os cálculos podem conduzir a uma solução com um erro relativo grande. Algumas alternativas foram sugeridas em |16| e em |19| para minimizar estes problemas e que são as seguintes:

- Alternativa 1: procurar em cada passo pelo elemento de maior magnitude na coluna do pivô (elementos da linha do pivô e aquelas abaixo) e trocar as linhas de tal modo que se possa usá-lo como elemento pivô. Um ciclo pode ser facilmente inscrito nos fluxogramas mostrados, antes da divisão por a_{kk}.

- Alternativa 2: procurar em toda a matriz de coeficientes pelo elemento de máxima magnitude. Em seguida trocar linhas e colunas até conseguir mover aquele elemento até a posição do pivô a_{11}. Após a eliminação dos elementos abaixo de a_{11} procurar na nova submatriz quadrada começando pela segunda linha e segunda coluna o novo elemento de máxima magnitude. O elemento encontrado é movido para a posição do novo pivô a_{22} e o processo é repetido. Como a mudança de colunas corresponde a mudanças na ordem das variáveis nas equações originais, é necessário manter um histórico para colocar as respostas finais na ordem original.

- Alternativa 3: dividir os elementos de cada linha no início por um fator de escala que depende da magnitude dos elementos da linha em questão. Algumas variações possíveis são: usar o elemento de maior magnitude daquela linha como divisor; usar a raiz quadrada do produto das magnitudes dos dois elementos de maior e de menor magnitude, ou seja, sua média geométrica. Após a divisão por um fator de escala de todas as linhas, é vantajoso repetir a operação para qualquer coluna onde todos os elementos se diferenciam por mais do que um fator limite, tal como 10^2, de elementos de outras colunas. Esta operação de escala representa uma mudança na variável original daquela coluna, e deve ser considerada nos resultados finais.

Vamos analisar um exemplo para melhor apreciar o que foi dito sobre tais erros de arredondamento e truncamento. Procuremos resolver um sistema pelo método de GAUSS usando nos cálculos 3 dígitos significativos.

$$\begin{aligned} x_1 + 3x_2 + 40x_3 &= 38 \\ 36x_1 + 106x_2 + 7x_3 &= -63 \\ 25x_1 + 5x_2 + 12x_3 &= 32 \end{aligned}$$

A solução exata de tal sistema é $x_1 = 1$; $x_2 = -1$; $x_3 = 1$

Inicialmente temos a seguinte matriz estendida:

$$\begin{bmatrix} 1 & 3 & 40 & 38 \\ 36 & 106 & 7 & -63 \\ 25 & 5 & 12 & 32 \end{bmatrix}$$

Após a primeira eliminação teremos:

$$\begin{bmatrix} 1 & 3 & 40 & 38 \\ 0 & -2 & -1430 & -1430 \\ 0 & -70 & -990 & -918 \end{bmatrix}$$

Após a segunda eliminação teremos:

$$\begin{bmatrix} 1 & 4 & 40 & 38 \\ 0 & 1 & 715 & 715 \\ 0 & 0 & 51100 & 49200 \end{bmatrix}$$

Após a terceira eliminação teremos:

$$\begin{bmatrix} 1 & 3 & 40 & 38 \\ 0 & 1 & 715 & 715 \\ 0 & 0 & 1 & 0{,}962 \end{bmatrix}$$

Utilizando a substituição ao contrário, teremos:

$$x_3 = 0{,}962$$

$$x_2 = 27$$

$$x_1 = -81{,}5$$

O que é, como se pode verificar, uma solução completamente errada. Se utilizarmos a alternativa número 1 teremos o seguinte:

- Iniciamos trocando a primeira e a segunda linhas e teremos:

$$\begin{bmatrix} 36 & 106 & 7 & -63 \\ 1 & 3 & 40 & 38 \\ 25 & 5 & 12 & 32 \end{bmatrix}$$

- Após a primeira eliminação teremos:

$$\begin{bmatrix} 1 & 2,94 & 0,194 & -1,75 \\ 0 & 0,06 & 39,8 & 39,8 \\ 0 & -72,5 & 7,1 & 75,8 \end{bmatrix}$$

- Trocando-se a segunda e a terceira linhas teremos:

$$\begin{bmatrix} 1 & 2,94 & 0,194 & -1,75 \\ 0 & -72,5 & 7,1 & 75,8 \\ 0 & 0,06 & 39,8 & 39,8 \end{bmatrix}$$

- Após a segunda eliminação teremos:

$$\begin{bmatrix} 1 & 2,94 & 0,194 & -1,75 \\ 0 & 1 & -0,0979 & -1,04 \\ 0 & 0 & 39,8 & 49,9 \end{bmatrix}$$

- Após a terceira eliminação teremos:

$$\begin{bmatrix} 1 & 2,94 & 0,194 & -1,75 \\ 0 & 1 & 0,0979 & -1,04 \\ 0 & 0 & 1 & 1,0 \end{bmatrix}$$

- Realizando a substituição ao contrãtio teremos:

$$x_3 = 1,0$$

$$x_2 = -0,94$$

$$x_3 = 0,82$$

Apesar de não ser uma solução exata ē uma aproximação muito melhor do que a encontrada anteriormente. Podemos então, com base neste exemplo, compreender que, apesar de que os computadores utilizam cerca de 8 dīgitos significativos ou atē mais, sō se procura utilizar computadores para solucionar sistemas com um nūmero de equações muito superior a três, como mostrado no exemplo, e para sistemas de porte maior o nūmero de algarismos significativos do computador usado pode não ser suficiente.

A aproximação da solução encontrada pode ser melhorada se alguns cuidados forem tomados. O vetor solução aproximado pode ser aplicado no sistema original e os novos valores das constantes podem ser avaliados assim;

$$0,82 + 3(-0,94) + 40(1) = 38,0$$

$$36(0,82) + 106(-0,94) + 7(1) = -63,1$$

$$25(0,82) + 5(-0,94) + 12(1) = 27,8$$

Podemos agora calcular os resīduos entre os valores originais das constantes e os valores aproximados obtidos assim:

$$\left| \begin{array}{l} r_1 = 38,0 - 38,0 = 0,0 \\ r_2 = -63 - (-63,1) = 0,1 \\ r_3 = 32 - 27,8 = 4,2 \end{array} \right.$$

As correções para os valores de $\underline{x_i}$ podem ser obtidas a partir da solução do novo sistema pelo mesmo método assim:

$$e_1 + 3e_2 + 40e_3 = 0,0$$

$$36e_1 + 106e_2 + 7e_3 = 0,1$$

$$25e_1 + 5e_2 + 12e_3 = 4,2$$

A solução do novo sistema fornece como solução: $e_1 = 0,170$; $e_2 = -0,0569$ e $e_3 = 0,0000160$, logo a nova solução corrigida é igual a:

$$x_1 = x_1 + e_1 = 0,820 + 0,170 = 0,990$$

$$x_2 = x_2 + e_2 = -0,940 - 0,0569 = -0,997$$

$x_3 = x_3 + e_3 = 1,0 + 0,000016 = 1,00$, que é muito mais próxima da solução real do que a anterior. Geralmente para uma ou duas aproximações é possível atingir uma precisão aceitável para a precisão desejada, em termos de número de algarismos significativos, muito embora possamos repetir o processo tantas vezes quanto o desejado para uma aplicação.

Uma comparação entre o método de GAUSS e o método de GAUSS-JORDAN é apresentado em |16| em termos de esforço computacional (EC) para a solução de um sistema qualquer, e apresenta os seguintes resultados:

- Método de GAUSS — $EC_1 = (n^3 + 3n^2 - n) / 3$
- Método de GAUSS-JORDAN — $EC_2 = (2/3)\ ((n^2 + 2n - 1)/(n^2 + n))$

onde $\underline{n}$ é o número de equações do sistema e tomou-se como base para os cálculos de EC o número de multiplicações efetuadas

em cada método.

Para sistemas com grande número de equações pode-se afirmar que o esforço computacional do Método de GAUSS é 2/3 do respectivo esforço do Método de GAUSS-JORDAN.

O método de GAUSS-JORDAN é com alguma freqüência utilizado para inversão de matrizes, após algumas adaptações.

Caso o sistema de equações seja singular, não existe inversa para a matriz de coeficiente. Isto pode ser compreendido pois as mesmas operações realizadas nas linhas para a inversão de uma qualquer matriz são as mesmas realizadas no Método de GAUSS-JORDAN para conseguir as eliminações. Se estas operações levarem a um termo nulo na diagonal, que não possa ser removido através de trocas de linhas, o sistema é dito singular e matriz inversa não existe.

A solução de um sistema de equações lineares simultâneas através de inversão e multiplicação de matrizes se torna muito interessante quando temos vários sistemas a resolver, todos possuindo a mesma matriz de coeficientes, mas vetores de constantes diferentes. É necessário calcular a matriz inversa apenas uma vez neste caso e em seguida usá-la como um pré-multiplicador para cada um dos vetores de constantes. O procedimento é o seguinte: para o sistema $AX = C$, pré-multiplicamos ambos os lados pela matriz inversa de A e temos $A^{-1} AX = A^{-1} C$. Aplicando a definição de matriz identidade ou matriz unitária, temos finalmente que $X = A^{-1} C$. Deste modo a multiplicação da inversa de A pelo vetor de constantes C fornece vetor coluna X com as solução do sistema pro

posto.

Os fluxogramas das figuras 4.1 e 4.2 podem ser adaptados para calcular a matriz inversa do seguinte modo, exposto como uma alteração na matriz de coeficientes, não pela inclusão do vetor de constantes mas pela inclusão de n colunas da matriz identidade ao lado direito da matriz de coeficientes. Cada nova linha agora passa a ter 2n elementos. Ao final das eliminações, quando a matriz de coeficientes foi convertida a uma matriz identidade, os coeficientes da matriz invertida ocuparão as posições que antes eram ocupadas pela matriz identidade. Um fluxograma para a inversão de uma matriz de n por n pelo método de GAUSS-JORDAN é visto na figura 4.4. O primeiro ciclo de comando DO cria a matriz extendida e o resto é similar ao procedimento mostrado anteriormente, com a exceção das alterações em índices onde apropriado. Um ciclo adicional pode ser adicionado, se desejado, para mover os elementos da matriz inversa para uma matriz B.

4.3 MÉTODOS ITERATIVOS

Após uma análise dos métodos diretos ou de eliminação na seção anterior, para a solução de sistemas de equações lineares simultâneas, passamos agora a considerar os métodos iterativos ou indiretos. Exatamente como no caso de métodos iterativos para cálculo de raízes de polinômios, aqui uma aproximação das incógnitas é dada inicialmente e em seguida, através de algumas técnicas que vamos estudar, procura-se chegar a uma melhor aproximação da solução desejada, repetindo-se o procedimento até que, no caso de haver convergên-

INÍCIO

$i = 1$

$j = n+1$

$(i+n) : j$

$\neq$ $a_{ij} = 0$

$a_{ij} = 1$

$j = j+1$

$j : 2n$ ($\leq$ / $>$)

$i = i+1$

$i : n$ ($\leq$ / $>$)

A

A

$K = 1$

B

$j = K$

$S = a_{KK}$

$a_{Kj} = \frac{a_{Kj}}{S}$

$j = j+1$

$j : 2n$ ($\leq$ / $>$)

$i = 1$

C

$i : K$ ($=$ E / $\neq$ D)

D

$j = K$

$S = a_{iK}$

$a_{ij} = a_{ij} - S a_{Kj}$

$j = j+1$

$j : 2n$ ($\leq$ / $>$)

E

$i = i+1$

$i : n$ ($\leq$ C / $>$)

$K = K+1$

$K : n$ ($\leq$ B / $>$)

FIM

Figura 4.4 – Inversão de matrizes pelo método de GAUSS – JORDAN. (Retirado de [19])

cia, a diferença entre duas aproximações sucessivas seja inferior do que uma precisão especificada por nós ao início do processo. No caso de sistemas de equações lineares, fornecemos no início, um conjunto de valores para X como primeira aproximação para a solução do sistema. Para que possamos prosseguir, é necessário que todas as incógnitas envolvidas tenham convergência satisfatória.

A filosofia básica dos métodos iterativos para a solução de sistemas de equações lineares é tentar escrevê-los da seguinte forma:

$$x = Fx + d$$

onde F é uma matriz de (n,n) e $\underline{d}$ é um vetor do R^n. A partir de uma aproximação inicial x_0, pertencente ao R^n, realizamos um conjunto de iterações do seguinte modo:

$$1) \quad x(1) = Fx(0) + d \ ,$$

$$2) \quad x(2) = Fx(1) + d \ ,$$

....

de tal modo que chegamos a uma fórmula de recorrência expressa por

$$x(k+1) = Fx(k) + d$$

Para o caso onde $\lim_{k \to \infty} x(k) - x = 0$, podemos dizer que a seqüência de aproximações de x(1) até x(k) converge para a solução desejada $\underline{x}$,

Algum cuidado deve ser tomado para o caso da seqüência não convergir, ou para decidir quando parar no caso de convergência.

Um critério que pode ser usado para decidir quando parar o processo em caso de convergência é comparar com uma certa tolerância, a diferença entre os valores sucessivos das aproximações. Quando a diferença for inferior à tolerência definida, podemos encerrar o processo.

Outra maneira é calcular o erro absoluto máximo entre duas iterações e compará-lo com uma tolerância previamente determinada.

A matriz F é comumente chamada de matriz de iteração e a convergência do processo depende de suas propriedades.

Passemos aos métodos iterativos propriamente ditos nas seções que seguem.

4.3.1 MÉTODO DE "JACOBI"

O Método de JACOBI também é conhecido como Método dos deslocamentos simultâneos (Method of simultaneous displacements) ou também como "Total step iteration" nos livros de Cálculo Numérico.

Inicialmente o sistema AX = C é rearranjado para que a primeira equação resolvida para x_1, a segunda para x_2, até chegarmos a última equação. Seguindo este passo temos:

$$x_1 = \frac{1}{a_{11}} (c_1 - a_{12}x_2 - a_{13}x_3 - \dots - a_{1n}x_n)$$

$$x_2 = \frac{1}{a_{22}} (c_2 - a_{21}x_1 - a_{22}s_3 - \dots - a_{2n}x_n)$$

....

$$x_n = \frac{1}{a_{nn}} (c_n - a_{1n}x_n - a_{2n}x_2 - \dots - a_{n,n-1}x_{n-1})$$

Analisando as resposta obtidas vemos que a matriz de iteração é a seguinte:

$$\begin{bmatrix} 0 & -a_{12}/a_{11} & -a_{13}/a_{11} \cdots & -a_{1n}/a_{11} \\ -a_{12}/a_{22} & 0 & -a_{23}/a_{22} \cdots & -a_{2n}/a_{22} \\ \cdots & \cdots & \cdots & \cdots \\ -a_{i1}/a_{ii} & -a_{i2}/a_{ii} & -a_{i3}/a_{ii} \cdots 0 \cdots & -a_{in}/a_{ii} \\ \cdots & \cdots & \cdots & \cdots \\ -a_{n1}/a_{nn} & -a_{n2}/a_{nn} & -a_{n3}/a_{nn} \cdots & 0 \end{bmatrix}$$

A matriz F de iteração obtida apresenta a característica de que $f_{ij}=0$ para $i=j$. Os demais elementos $f_{ij}=a_{ij}/a_{ii}$ para $i \neq j$. O vetor d quando expressamos as iterações por $x=Fx + d$ é formado por c_1/a_{11}, c_2/a_{22}, ..., c_n/a_{nn}. De uma forma geral podemos expressar as iterações do seguinte modo:

$$x_1(k+1) = \frac{1}{a_{11}} \left(c_1 - a_{12}x_2(k) - a_{13}x_3(k) - \dots - a_{1n}x_n(k)\right)$$

$$x_2(k+1) = \frac{1}{a_{22}} \left(c_2 - a_{21}x_1(k) - a_{23}x_3(k) - \dots - a_{2n}x_n(k)\right)$$

....

$$x_n(k+1) = \frac{1}{a_{nn}} \left(c_n - a_{n1}x_1(k) - a_{n2}x_2(k) - \dots - a_{n,n-1}x_{n-1}(k)\right)$$

Os índices $\underline{k}$ e $\underline{k+1}$ que aparecem entre parênteses indicam o número da iteração e servem também para diferenciar os conjuntos de valores de $\underline{x}$ obtidos ao longo das várias iterações. O processo continua até que a diferença entre os valores de duas iterações consecutivas esteja dentro de um limite pré-estabelecido. Como já foi dito antes, existem vários modos de se fazer isto. Além daqueles já mencionados no início da seção 4.3, existe um que é obtido através da soma dos valores absolutos dos desvios relativos entre todas as iterações já realizadas e compará-las com um desvio permitido e já fixado. Isto pode ser expresso por:

$$\frac{x_1(k+1) - x_1(k)}{x_1(k+1)} + \frac{x_2(k+1) - x_2(k)}{x_2(k+1)} + \dots + \frac{x_n(k+1) - x_n(k)}{x_n(k+1)} = S \geqslant 0$$

e para a soma calculada S temos S $\leqslant$ desvio permitido.

Como se pode observar o Método de JACOBI é muito fácil de ser programado e apenas alguns cuidados devem ser tomados para decidir quando interromper o processo, por ter sido observado que o mesmo é divergente ou porque chegamos a uma precisão considerada como aceitável para nossa aplicação específica.

4.3.2 MÉTODO DE GAUSS-SEIDEL

Este método pode ser considerado como uma variante refinada do método anteriormente discutido. Ao invés de usar todos o cojunto de valores obtidos para $\underline{x}$ na iteração anterior, é de se esperar que seja vantajoso utilizar o valor mais recente de cada variável calculada no passo anterior. Podemos então expressar as iterações de um modo geral assim:

$$x_1(k+1) = \frac{1}{a_{11}} (c_1 - a_{12}x_2(k) - a_{13}x_3(k) - \ldots - a_{1n}x_n(k))$$

$$x_2(k+1) = \frac{1}{a_{22}} (c_2 - a_{21}x_1(k+1) - a_{23}x_3(k) - \ldots - a_{2n}x_n(k))$$

....

$$x_n(k+1) = \frac{1}{a_{nn}} (c_n - a_{n1}x_1(k+1) - a_{n2}x_2(k+1) - \ldots - a_{n,n-1}x_{n-1}(k+1))$$

O Método de GAUSS-SEIDEL é mostrado através de fluxograma na figura 4.5 onde foram colocados testes apropriados para encerrar o processamento em caso de divergência ou de se chegar a uma solução aceitável.

Apesar de não incluído no fluxograma mostrado na figura 4.5, tanto neste método quanto no anterior, as equações deveriam ser arranjadas de tal modo que o coeficiente de x_1 de maior magnitude ocorra na primeira equação, o coeficiente de x_2 de maior magnitude ocorra na segunda equação e o mesmo ocorra nas demais equações. Esta recomendação se deve ao fato de afetar a convergência de ambos os métodos apresentados e um estudo mais profundo pode ser visto em |19|.

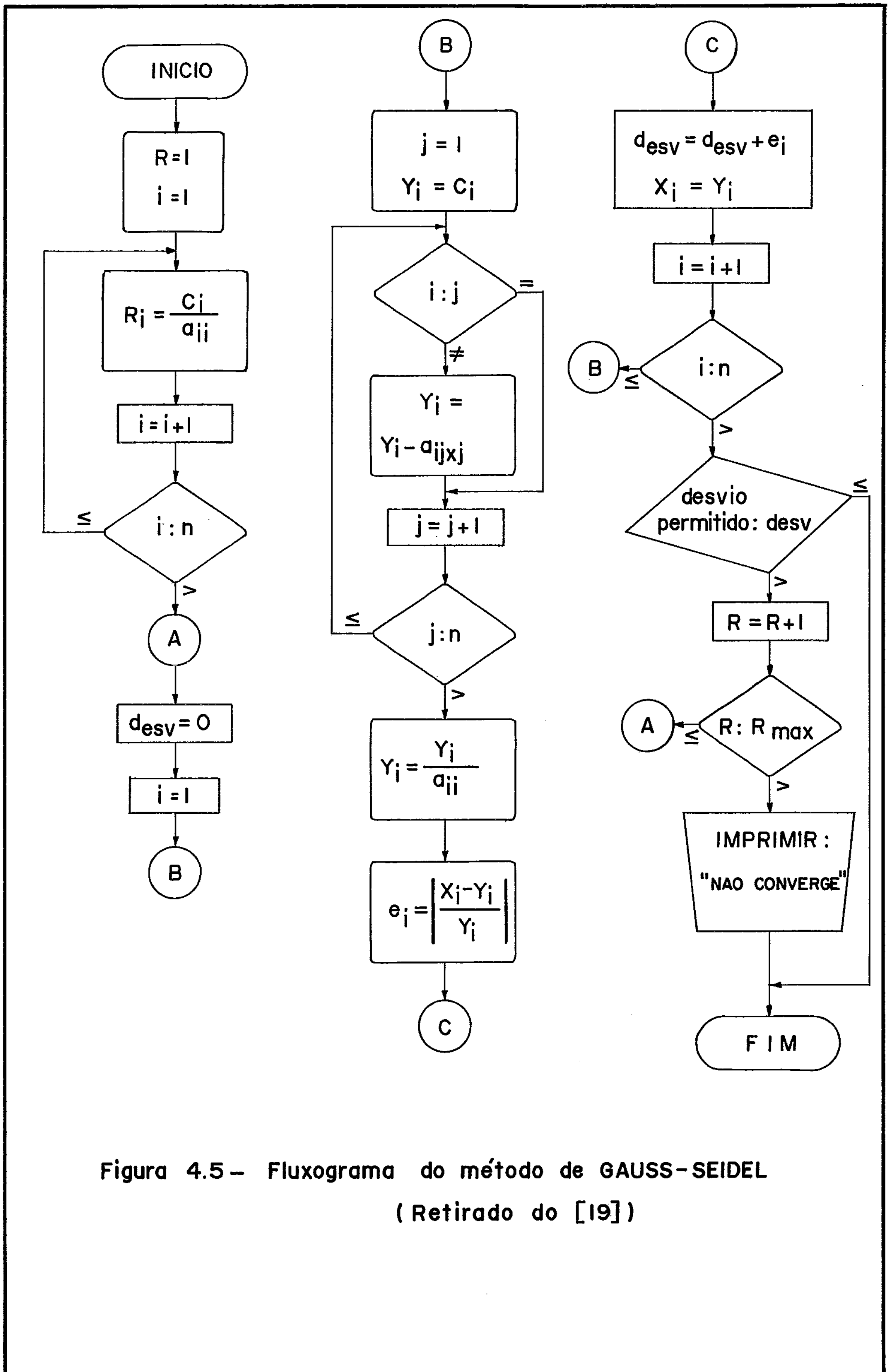


Figura 4.5 – Fluxograma do método de GAUSS-SEIDEL (Retirado do [19])

4.3.3 PROBLEMAS DE CONVERÊNCIA DOS MÉTODOS

O método inicialmente analisado, o de JACOBI, apresenta a seguinte matriz de iteração:

$$\begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & & 0 \end{bmatrix} = (L + U)$$

Se considerarmos a matriz diagonal D, a sua inversa será D^{-1} teremos:

$$\begin{bmatrix} 0 & -a_{12}/a_{11} & \cdots & -a_{1n}/a_{11} \\ -a_{21}/a_{22} & 0 & \cdots & -a_{2n}/a_{22} \\ \cdots & \cdots & \cdots & \cdots \\ -a_{n1}/a_{nn} & -a_{n2}/a_{nn} & \cdots & 0 \end{bmatrix}$$

Esta última matriz, a matriz de iteração do método de JACOBI pode ser expressa por:

$$F = -D^{-1} (L + U)$$

da equação $x = Fx + D$, e temos também que $d = D^{-1}c$, onde c representa o vetor de constantes do sistema.

O Método de GAUSS-SEIDEL apresenta a seguinte matriz de iteração:

$$F = -(D + L)^{-1} U$$

e o vetor d temos a seguinte estrutura:

$$d = (D + L)^{-1} c$$

Estas considerações iniciais sobre matrizes de iteração foram apresentadas pois no estudo de convergência dos métodos, elas têm uma importância fundamental, pois delas depende a convergência ou não de um dos métodos.

Para um sistema modificado e expresso na forma $x = Fx + d$, onde as iterações são definidas por:

$$x(k+1) = Fx(k) + d, \quad \text{para } k = 1,2, \ldots.$$

existem algumas condições que são suficientes para sua convergência e que são:

1) $\sum_{j=1}^{n} |f_{ij}| < 1$ para todo i, (i=1, 2, ..., n)

2) $\sum_{i=1}^{n} |f_{ij}| < 1$ para todo j, (i=1, 2, ..., n)

3) $\sum_{\substack{i=1 \\ j=1}}^{n} |f_{ij}|^2 < 1$

Estas condições e outras são mostradas em |16| e em |19| com maiores detalhes e ali também dadas novas referências.

Para que os métodos apresentados possam convergir, é condição suficiente também que a matriz do sistema seja diagonalmente dominante como proposto em 4.2.3 e irredutível para qualquer escolha do vetor inicial x(0) e para qualquer vetor C. Neste caso, o Método de GAUSS-SEIDEL converge mais rapidamente do que o Método de JACOBI. Um cuidado deve ser tomado quando comparamos os primeiros resultados de ambos os métodos pois não se pode garantir que os resultados obtidos nas primeiras iterações do Método de GAUSS-SEIDEL sejam melhores do que aquelas observadas no Método de JACOBI.

4.4 CONSIDERAÇÕES SOBRE OS MÉTODOS

Cada uma das técnicas consideradas, de eliminação e iterativas, possuem vantagens e desvantagens que procuramos sumarizar aqui:

a) Os métodos de eliminação apresentam a vantagem de poderem sempre ser aplicados. O mesmo não se pode dizer dos métodos iterativos que dependem da convergência. Os métodos de eliminação possuem a característica de que utilizamos um número fixo de operações para um determinado número de equações, não importando quais sejam os números envolvidos.

b) É possível detetar a singularidade de um sistema nos métodos de eliminação. O mesmo não acontece com os métodos iterativos.

Um qualquer sistema em questão, pode ser indeterminado ou impossível e não percebemos isto quando utilizamos um método iterativo.

c) Os métodos iterativos são menos afetados pelos erros de arredondamento do que os métodos diretos. Nos métodos de eliminação os coeficientes são alterados periodicamente e o mesmo não acontece nos métodos iterativos. Isto pode ser minimizado pela reorganização das equações de modo a manter os elementos de maior magnitude na diagonal mas, mesmo assim, isto representa um sério problema no manuseio de grandes matrizes.

d) Para os sistemas esparsos, onde existem um grande número de coeficientes nulos, o número de operações é grandemente reduzido, e os métodos iterativos apresentam uma grande vantagem sobre os métodos de eliminação.

Para os sistemas de grande porte seria mais recomendável trabalhar com métodos iterativos, desde que haja convergência, pois os sistemas podem ter sua solução completamente distorcida pelos erros de arredondamento e truncamento dos métodos de eliminação.

Agora que analisamos os principais pontos considerados fundamentais para o escopo deste trabalho, (nos capítulos de um até o quatro), e que concluímos que os sistemas iterativos são os mais adequados para grandes sistemas por introduzirem menor quantidade de erros de arredondamento e de truncamento, além de serem os mais apropriados para sistemas esparsos, passemos ao capítulo seguinte onde os sistemas esparsos serão analizados com maior detalhe.

5. MATRIZES ESPARSAS

As matrizes esparsas aparecem em diversos tipos de problemas como: na análise de estruturas reticuladas ou contínuas pelo método dos elementos finitos; na aplicação do método de Newton à equações algébricas não lineares; na análise de sistemas de força e de redes elétricas; no projeto de circuitos integrados de computadores e muitos outros.

Por ser o ponto central desta pesquisa, inicialmente vamos caracterizar melhor o que são matrizes esparsas e os problemas que normalmente são encontrados em sua manipulação quando se desenvolvem aplicações em que elas ocorrem. Em seguida, vamos analisar as principais alternativas de armazenamento sugeridas por diversos autores e apontar suas principais vantagens e desvantagens. Com base nesta análise passamos em seguida a analisar o impacto da característica de esparsidade das matrizes sobre os principais algorítmos conhecidos para manipulação de matrizes. Apresentamos também como foram realizados os testes e os algorítmos usados no B6700 do NCE/UFRJ escritos em FORTRAN e finalizamos este capítulo discutindo os problemas encontrados na implantação de tais algorítmos de tal modo que os leitores possam utilizar o conhecimento adquirido ao longo deste trabalho para adaptarem, se for o caso, os algorítmos e as idéias para outras linguagens e outros equipamentos disponíveis em suas instalações de origem.

5.1 O QUE SÃO MATRIZES ESPARSAS

Uma matriz será dita esparsa quando apresentar uma pequena percentagem de elementos não nulos. Outra forma de apresentar esta idéia pode ser compreendida no caso de termos uma matriz quadrada A de ordem $\underline{n}$ com $\underline{r}$ elementos não nulos. A matriz A será dita esparsa se tivermos $r \ll n^2$. O interesse em formular soluções para problemas de tratamento de matrizes esparsas tem larga aplicação, como já foi dito, e geralmente problemas importantes não podem ser resolvidos porque conduzem a grandes matrizes que não podem ser invertidas por falta de memória ou porque tais processos seriam caros demais. Como em muitos casos estas matrizes são esparsas, é muito importante conhecer as técnicas existentes para tratá-las. Dependendo de cada aplicação e de suas características específicas poderá o leitor escolher a que mais lhe aprouver.

Chamaremos, neste trabalho, de Índice de Esparsidade a percentagem de elementos nulos dentro da matriz. Durante as operações realizadas para solucionar sistemas de equações, o índice de esparsidade de uma matriz geralmente varia. Este índice pode ser um excelente indicador da eficiência dos critérios adotados para armazenar a matriz em questão e também do algorítmo usado para resolver o problema. A característica de esparsidade das matrizes envolvidas deve ser aproveitada e a filosofia básica é a de evitar o armazenamento e operação com todos os coeficientes nulos ou com grande parte deles. Os algorítmos projetados para matrizes esparsas devem se preocupar com a esparsidade das mesmas

durante as transformações. Porém uma das grandes dificuldades é justamente manter esta característica, pois muitas operações com coeficientes nulos acabam transformando-os em não nulos. Isto é particularmente acentuado nos métodos diretos de resolução de sistemas de equações. Em métodos iterativos este índice não se altera durante as iterações sucessivas.

Quando o usuário trabalha com um tipo especial de matriz esparsa, ele pode explorar suas propriedades especiais e introduzí-las nos seus algorítmos. Quando não se conhece à priori o tipo de matriz esparsa que vamos lidar, temos que construir algorítmos que sejam gerais mas que incorporem as idéias estudadas em capítulos anteriores assim como as que serão discutidas nesta parte do trabalho. Todas as análises apresentadas daqui em diante serão para grandes matrizes esparsas em um equipamento que utilize memória virtual. Outra hipótese de trabalho é a de que não conhecemos à priori as características da matriz esparsa e nossos algorítmos serão gerais, isto é, não serão feitos especialmente para explorar propriedades particulares de determinadas formas de matrizes esparsas. O leitor poderá introduzir tais melhoramentos se considerar necessário em suas próprias aplicações.

Em muitos tipos de problemas surgem matrizes com um número significante de elementos nulos. Exemplos comuns são a matriz diagonal, que pode ser considerada como um vetor, e a matriz triangular, onde basta especificar a parte superior ou inferior conforme o caso em questão. Em tais casos é interessante encontrar uma forma de armazenamento que não necessite de espaço para os elementos nulos. Para o caso

da matriz diagonal a solução é trivial, bastando utilizar-se um vetor de n componentes para armazenar uma matriz diagonal de ordem n. Para o caso de uma matriz triangular, inferior por exemplo, isto pode ser feito através de um vetor de $n(n+1)/2$ elementos, onde cada elemento da matriz original pode ser recurperado através do seguinte artifício de cálculo:

$$\hat{d}(i,j) = \hat{d}(1,1) + \frac{i(i-1)}{2} + (j-1)$$

onde, $1 \leq i \leq n$, $1 \leq j \leq i$ e também $d(i,j)$ representa o endereço do elemento (i,j) da matriz original e $d(1,1)$ é o endereço do elemento $(1,1)$ da matriz original. Outro tipo também comum de matriz com um grande número de elementos nulos é a matriz tridiagonal onde as seguintes propriedades são válidas para um elemento (i,j) de uma matriz de ordem n: $1 \leq i \leq n$; $j=1,2$ para $i=1$; $j=i-1, i, i+1$ para $i \leq i \leq n$, e $j=n-1, n$ para $i=n$. Para armazenar tal matriz podemos usar um vetor onde cada elemento pode ser recuperado através da seguinte fórmula de cálculo

$$\hat{d}(i,j) = \hat{d}(1,1) + 2(i-1) + j - 1$$

O vetor teria 3n-2 elementos não nulos. Outra forma de armazenar uma matriz tridiagonal é considerá-la como formada por três vetores de n componentes, onde os elementos são definidos por $d(i+1,i)$, $d(i,i)$, e $d(i,i+1)$. Para cada um dos vetores mencionados definiríamos uma fórmula para recuperação de elementos da matriz original. Porém nas aplicações práticas surgem também diversos outros tipos de matrizes com um grande

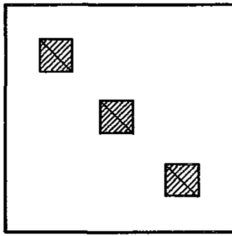
número de elementos nulos. Tewarson |1| apresenta 10 formas canônicas de matrizes esparsas em sua obra, formas essas que não são tão triviais como as que foram apresentadas em termos de se encontrar uma forma de armazenamento e de recuperação para seus elementos não nulos. Cada uma das matrizes mencionadas constituem casos particulares de matrizes blocadas de caráter mais geral. Por exemplo, se considerarmos uma matriz diagonal de ordem n, ela é um caso particular de uma matriz diagonal blocada com a seguinte forma:

$$\begin{bmatrix} D1 & 0 \;\ldots\ldots 0 \\ 0 & D2 \ldots\ldots 0 \\ \ldots\ldots & \ldots\ldots\ldots \\ 0 & 0 \;\ldots\ldots Dn \end{bmatrix}$$

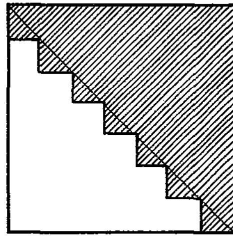
onde D1, D2, ..., Dn são matrizes quadradas e os zeros representam matrizes cujos componentes são todos nulos. Alguns tipos de matrizes apresentadas em |1| podem ser vistas na figura 5.1. No caso mais geral então, se N é o número de elementos não nulos de uma matriz D de dimensões $(i_1, i_2, \ldots, i_s)$, onde $1<i_1<n_1, \ldots, 1<i_s<n_s$, a matriz será dita esparsa se:

$$N \ll \prod_{i=1}^{s} n_i$$

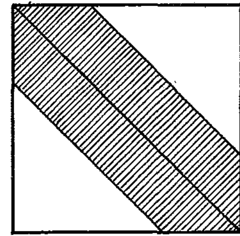
Uma matriz pode apresentar um número razoável de elementos nulos e não ser esparsa, como por exemplo a matriz triangular. Uma matriz que possua um número significativo de elementos nulos pode ser convertida a uma das formas canônicas através da permutação de linhas e colunas. A operação



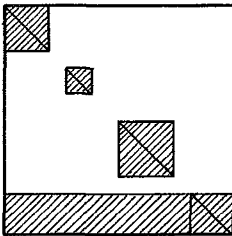
A) Diagonal Blocada (BDF)



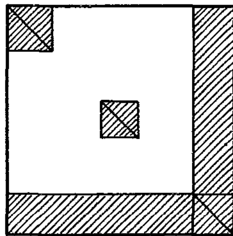
B) Triangular Blocada (BTF)



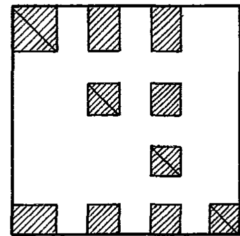
C) Matriz de Banda (BF)



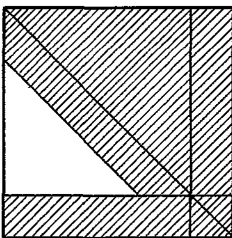
D) Diagonal Blocada limitada (SBBDF)



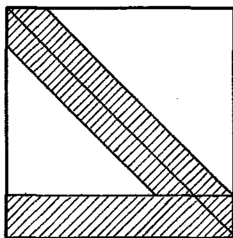
E) Diagonal Blocada duplamente limitada (DBBDF)



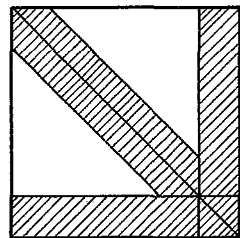
F) Triangular Blocada e limitada (BBTF)



G) Matriz de Banda triangular blocada (BBNTF)



H) Matriz de Banda limitada (SBBF)



I) Matriz de Banda duplamente limitada (DBBF)

Figura 5.1 – Matrizes blocadas típicas (Retirado de [1])

de reduzir uma matriz a uma das formas canônicas é um problema combinatório |1|. A escolha da forma canônica mais adequada para um certo caso depende do conhecimento maior ou menor sobre a aplicação em estudo e não existe uma receita para solucionar tal questão. Passemos em seguida a analisar as diversas formas de armazenar matrizes esparsas. O ideal seria conseguir um meio de minimizar tanto a área total usada e o tempo total de processamento. De um modo geral, estes requisitos, área de memória mínima e tempo total mínimo são incompatíveis e uma solução de compromisso deve ser tentada.

5.2 ALTERNATIVAS DE ARMAZENAMENTO

Após as considerações feitas sobre a necessidade de se procurar meios para armazenar matrizes esparsas que minimizem o uso de memória principal e que reduzam, por conseqüência, o tempo de processamento em virtude do tratamento apenas dos elementos não-nulos, passamos agora a tratar especificamente do problema de armazenamento.

Grandes matrizes esparsas são geralmente armazenadas de modo comprimido, ou seja, apenas os elementos não nulos são armazenados assim como algumas informações de controle que permitam sua recuperação quando necessário. Tewarson |1| apresenta quatro razões básicas para o uso de armazenamento de forma comprimida e que são:

a) Maiores matrizes podem ser armazenadas e tratadas na memória principal, o que seria impossível de outro modo.

b) Existem casos em que mesmo em forma comprimida as matrizes não cabem na memória principal, como no caso de sistemas de "time sharing" e é necessário se utilizar memória auxiliar como discos ou fitas. As operações de transferência de dados de tais meios externos para a memória principal são operações lentas quando comparadas com operações na memória principal. Isto é mais uma razão para se usar a forma comprimida de armazenamento para memória auxiliar, para redução do tempo de entrada e saída e redução do uso de memória auxiliar.

c) Uma redução substancial do tempo de processamento pode ser conseguida se as operações com elementos nulos não forem realizadas. Isto é conseguido através de um processamento simbólico onde apenas operações triviais como testes são realizadas.

d) A inversa de uma matriz expressa como resultante do produto de duas matrizes elementares (onde somente os elementos não triviais são armazenados em forma comprimida), geralmente necessita de menos memória do que a inversa explícita de uma matriz, mesmo que esteja armazenada de modo comprimido.

Em diversos algorítmos preparados para transformar uma certa matriz de uma forma para outra desejada, elementos adicionais não nulos são criados ao longo dos vários passos do procedimento. Deste modo os algorítmos para tratar esta forma comprimida de armazenamento devem estar preparados para a situação em elementos novos e não nulos devem ser inseridos nas linhas ou colunas, à medida que os passos

do procedimento são executados.

Em qualquer das alternativas discutidas neste trabalho, ou em outras projetadas pelo leitor, seria interessante notar que para valer a pena o esforço computacional de compactar as informações de controle e os valores propriamente ditos em um determinado modo, a condição desejável para o problema de armazenamento é que a área utilizada seja substancialmente menor que a área total que a matriz ocuparia se estivesse descompactada, ou seja, se utilizarmos 3 posições para armazenar cada elemento A(I,J) de uma matriz A de $\underline{k}$ elementos não nulos, então a área total será de 3k elementos. Para que haja vantagens no procedimento de compactação devemos ter 3k muito menor do que n^2 se a matriz A é de ordem $\underline{n}$. Outro modo de dizer isto é que existe um certo limite para o qual o uso do método não é aconselhável pois ocupa tanta memória quanto armazenar a própria matriz de forma descompactada.

Para solucionar tais problemas seria desejável encontrar um método que otimizasse o uso de memória e de tempo de processamento dos programas ao mesmo tempo. Tal alternativa parece não existir e como tal uma solução, pelo menos parcial, deve ser tentada.

5.2.1 USO DE LISTAS LINKADAS

O uso de listas linkadas para armazenamento de matrizes esparsas pode ser encontrado em |1| e em |10| de forma mais ampla e didática. Procuramos apresentar aqui esta alternativa de modo compacto, sem, no entanto, omitir suas prin-

cipais propriedades.

Consideremos uma matriz A onde cada elemento é referenciado por A(I,J). Por hipótese a matriz possui k elementos não nulos e é de ordem n. Cada elemento da matriz é armazenado na tripla (i,a,p), onde i é o subscrito da linha, a é o valor do elemento A(I,J) e p é o endereço próximo do elemento não nulo da coluna j. Cada elemento A(I,J) é armazenado como um elemento de sua coluna j, ou seja, o armazenamento é feito por coluna. Isto pode ser visto na figura 5.2 apresentada a seguir.

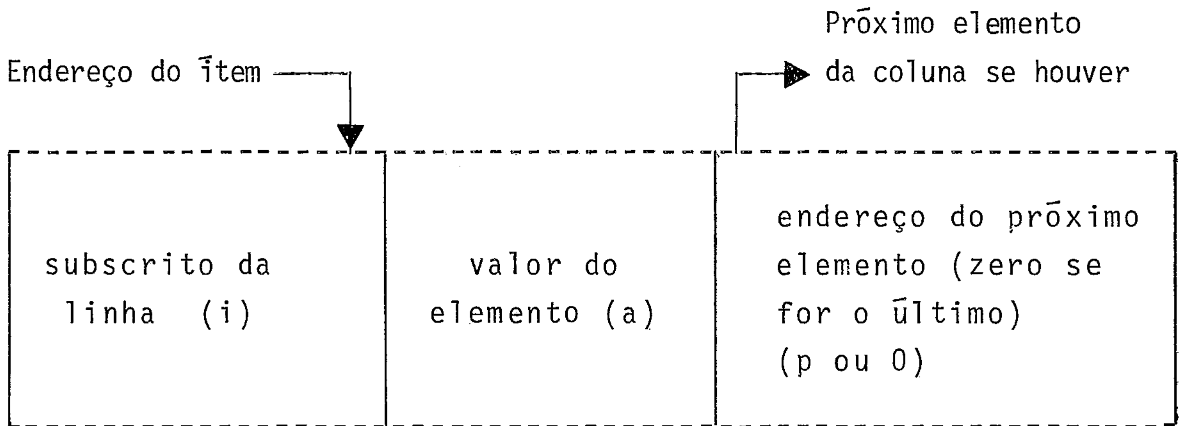


Figura 5.2 Estrutura de cada elemento da lista linkada (Retirado de |1|)

A estrutura da lista proposta é a seguinte: Uma área reservada para armazenar o endereço de início do primeiro elemento não nulo de cada coluna é criada e mantida ao longo das operações com a matriz. Esta área tem n posições contíguas. Outra área que também faz parte da lista é uma área para armazenamento das triplas como definidas anteriormente. Como existem k elementos não nulos na matriz A, utilizaremos uma área total de n+3k elementos para armazená-la.

O procedimento de recuperação de qualquer elemento dentro da lista é muito simples. Chamemos a primeira série de "área de endereços" e a segunda de "área de armazenamento". A área de endereços é contígua e cada posição aponta para o primeiro elemento não nulo da coluna respectiva. Por exemplo, a j-ésima posição da área de endereços possui como conteúdo o endereço do primeiro elemento não nulo da coluna j da matriz original. O elemento apontado está na área de armazenamento e ocupa 3 posições que são os componentes da tripla (i,a,p) definida anteriormente. As triplas armazenadas nesta área não necessitam ser contíguas, visto que são todas do mesmo comprimento e possuem um apontador para a próxima tripla. A principal vantagem deste método é que durante os cálculos podem aparecer elementos não nulos e o armazenamento de novas triplas se faz apenas alterando os conteúdos de apontadores relacionados, sem a necessidade de movimentação das triplas já existentes para uma operação de inserção no meio da lista. Igualmente, se durante os cálculos algum elemento se tornar nulo, a área ocupada pela tripla correspondente pode ser liberada e quando necessário poderá ser usada para armazenar outra qualquer tripla. Outra necessidade será a de manter uma lista de área disponível para evitar problemas de localizar um espaço disponível para guardar uma tripla. Em resumo, teremos duas listas: uma de área em uso e outra de área disponível para utilizarmos esta alternativa como descrito em |1|. Para o caso de uma matriz A de 500 por 500, onde existam três elementos não nulos por linha teremos a seguinte situação:

a) Área total que seria utilizada = 250.000 posições.

b) Elementos não nulos = 1500.

c) Área total compactada = 500 + 3 * 1500 = 5.000 posições.

d) Percentagem da área total utilizada = 2%.

e) Índice de esparsidade = 99,40%

Nesta situação o método conduziria a uma redução do uso de memória realmente substancial.

Analisemos outra situação onde uma matriz A de ordem 500 possua, por exemplo, 166 elementos não nulos por linha. Teremos o seguinte quadro:

a) Área total que seria utilizada = 250.000 posições.

b) Elementos não nulos = 166 * 500 = 83.000

c) Área total compactada = 500 + 3 * 83.000 = 249.500 posições.

d) Percentagem da área total utilizada = 99,80%.

e) Índice de esparsidade = 66.80%.

Observando os dois casos mostrados, vemos que até um índice de esparsidade de 66% este método pode ser vantajoso, porém, abaixo de 66% certamente que este método conduzirá a um gasto excessivo de memória principal e talvez seja mais adequado algum outro método apresentado neste trabalho. É importante lembrar que chamamos de Índice de Esparsidade de uma matriz, neste trabalho, a percentagem de elementos nulos existentes na matriz.

Podemos então, com base nesta análise, ter uma boa idéia das vantagens deste método, em termos de uso de memória e de facilidade de inserção e retirada de elementos da

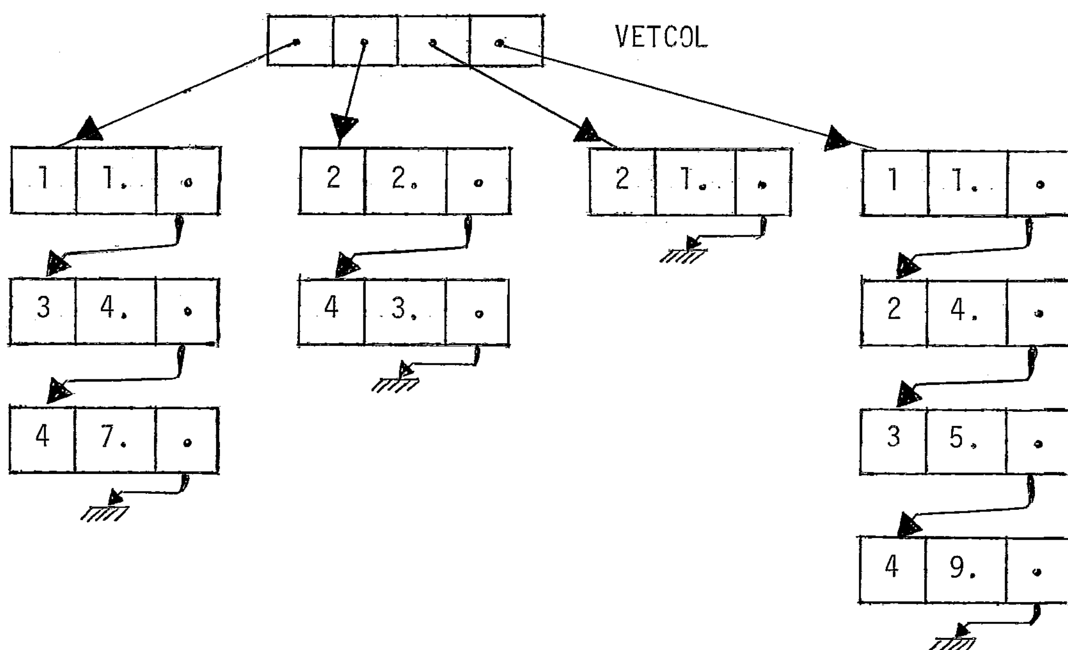
estrutura apresentada, através da manipulação de apontadores. Esta segunda característica do método deve ser levada em conta em virtude do aparecimento de elementos não nulos que geralmente surgem em muitas operações básicas com matrizes. Para conseguirmos este objetivo devemos manter o controle sobre duas listas, uma de áreas em uso e outra de áreas disponíveis.

5.2.1.1 ASPECTOS COMPUTACIONAIS

Para realizarmos algumas outras considerações sob o ponto de vista computacional, tomemos a matriz esparsa a seguir como exemplo:

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

Utilizemos a alternativa proposta através do uso de listas linkadas, onde cada lista representa os elementos não nulos da coluna $\underline{i}$ e é endereçada pelo apontador VETCOL (I). Consideremos então a representação gráfica das listas que representam a implementação da matriz A, do exemplo:



A lista linkada foi estruturada criando-se um vetor VETCOL de apontadores e de uma matriz XINFO que contem as seguintes informações:

XINFO (J,1) contém o subscrito da linha.

XINFO (J,2) contém o valor do elemento não nulo.

CINFO (J,3) contém o apontador ou endereço do próximo elemento não nulo da coluna. Conterá o valor zero se for o último.

Agora que já temos uma compreensão maior de como pode ser feita a implementação de tais idéias em FORTRAN, vejamos como podemos operar as matrizes segundo estas definições. Para exemplificar, consideremos o produto escalar como um procedimento básico para nossas análises. Inicialmente desenvolvemos um algorítmo segundo os critérios tradicionais armazenando toda a matriz A, e realizamos diversas medições que serão mostradas a seguir. Desenvolvemos, também, outro

algorītmo usando a filosofia de listas linkadas para realizar a mesma operação de produto escalar. O algorītmo tradicional tinha a seguinte estrutura básica em FORTRAN:

```
...........
ILIMIT = MAX + 1
XPROD = 0.
I = 1
IF (I .EQ. ILIMIT) GO TO 20
XPROD = XPROD + CONST (I) * A (I, ICOL)
I = I + 1
GO TO 10
CONTINUE
...........
```

Procuramos, em seguida, testar o novo algorītmo que possuīa a seguinte estrutura básica em FORTRAN igualmente:

```
...........
XPROD = 0.
PONTEI = VETCOL (ICOL)
IF (PONTEI .EQ. 0) GO TO 20
      XPROD = XPROD + CONST(IFIX(PONTEI,1))) *
              * XINFO (PONTEI, 2)
      PONTEI = IFIX (XINFO (PONTEI, 3))
      GO TO 10
CONTINUE
...........
```

Para que possamos comparar os dois algorítmos apresentados consideremos que a matriz A possui uma densidade média por coluna igual a k e que vamos considerar o número de operações necessárias aos cálculos, em ambos os casos, assim como o tempo de CPU do B6700 em milisegundos para o compilador FORTRAN utilizado, e conseguimos, assim, o seguinte quadro:

OPERAÇÕES REALIZADAS EM FORTRAN/B6700	USANDO LISTAS LINKADAS		MÉTODO TRADICIONAL	
	NÚMERO DE OPERAÇÕES	TEMPO DE CPU (mseg)	NÚMERO DE OPERAÇÕES	TEMPO DE CPU (mseg)
Adições	kM	kM(3,18)	2M+1	(2M+1)(3,18)
Produtos	kM	kM(6,03)	M	6,03M
Consulta do Vetor	1+kM	(1+kM)4,10	M	4,10M
Consulta a matriz	3kM	(3kM)12,30	M	12,30M
IF (cond) GO TO	kM	5,20kM	M+1	(M+1)5,20
GO TO	kM	2,0kM	M	2,0M
Inicializações reais	1	2,2	1	2,2
Inicializações inteiras	0	0,0	1	2,0
T O T A L		57,41kM+6,3		35,99M+12,58

Observamos que à medida que a ordem das matrizes (M) utilizadas nos testes aumentava, a relação entre os tempos de CPU se comportava do seguinte modo:

Tempo de CPU usando listas/tempo de CPU tradicional =
= 57.41k/35.99, onde k representa a densidade da matriz

considerada. Para os testes foi usada uma matriz de 10% de densidade e observamos que o teste feito em 20 corridas permitiu realizar o produto escalar usando listas em aproximadamente 16% do tempo gasto utilizando-se o processo tradicional.

Após estas considerações sobre o problema computacional desta alternativa, passemos a analisar outros métodos possíveis para compressão dos dados, a conseqüente redução de memória principal necessária e possivelmente uma redução do tempo de CPU, como observamos neste caso. As outras alternativas a serem estudadas não se baseiam em listas linkadas.

5.2.2 OUTROS MÉTODOS DE COMPRESSÃO

O tratamento de grandes matrizes esparsas pode conduzir à situações onde o método anterior não seja adequado, principalmente se a disponibilidade de memória principal for pequena quando comparada com as dimensões da matriz em questão. Os métodos de compressão que vamos estudar agora, utilizam menos memória, porém têm o grande problema de apresentarem uma relativa dificuldade para resolver as situações de inserções e retiradas de elementos durante o processamento. Operações de movimentação física de dados são freqüentemente utilizadas na memória e isto pode ser uma das razões que conduzam a um aumento substancial para o aumento do tempo de processamento. Existem casos onde somente parte da matriz pode ser carregada na memória principal e então teremos que lidar com o problema do aumento do tempo de entrada e saída.

5.2.2.1 ARMAZENAMENTO COMPRIMIDO EM VETOR

Neste método cada elemento não nulo da matriz esparsa utiliza duas posições do vetor. A primeira posição contém o subscrito da linha e a segunda posição contem o valor do elemento. O armazenamento é feito por colunas. Para indicar o fim de uma coluna utiliza-se um valor zero no subscrito da linha. Para indicar qual a próxima coluna utiliza-se a segunda posição. Outro cuidado é a indicação de fim da matriz que pode ser conseguido através da colocação de, por exemplo, dois valores zeros seguidos. Este método utiliza para matriz A de ordem n e de k elementos não nulos,(n + k + 1)*2 posições de memória.

Analizemos o exemplo de uma matriz A de ordem 200 e que possua 7 elementos não nulos a saber: a_{21}, a_{41}, a_{52}, a_{13}, a_{33}, a_{24} e a_{45}. O vetor resultante teria a seguinte estrutura:

$(0,1;2,a_{21};4,a_{41};0,2;5,a_{52};0,3;1,a_{13};3,a_{33};0,4;2,a_{24};0,5;4,a_{45};0,0)$.

O vetor resultante apresentou informações de controle para as cinco colunas que possuem elementos não nulos, um indicador de fim da matriz e também informações sobre os sete elementos não nulos, todas as demais colunas que não possuem elementos não nulos, não necessitam ser mencionadas neste processo.

Para o exemplo mostrado temos as seguintes medidas:

a) Total de elementos da matriz = 40.000 (40.000 posições).

b) Elementos não nulos = 7.

c) Área total usada = (5+7+1)*2 = 26 posições.

d) Percentagem da área total utilizada = 0.0175% e

e) Índice de esparsidade = 99.98%

Suponhamos outro exemplo onde temos uma Matriz A de ordem 200,onde existem 98 elementos não nulos por coluna. Para esta situação teríamos as seguintes medidas:

a) Total de elementos da matriz = 40.000 (40.000 posições).

b) Elementos não nulos = 98 * 200 = 19.600.

c) Área total compactada = (200 + 19.600 + 1) * 2 = = 39.602 posições.

d) Percentagem da área total utilizada = 99% e

e) Índice de esparsidade = 51%

Podemos concluir destes dois exemplos apresentados que este método pode ser usado com vantagens para matrizes com um índice de esparsidade maior ou igual a 51%.

5.2.2.2 ARMAZENAMENTO COMPRIMIDO COM TRÊS VETORES

Este método utiliza três vetores para armazenar uma matriz esparsa qualquer a saber: um vetor para armazenar os valores dos elementos não nulos; um vetor para armazenar os subscritos das linhas respectivas e outro vetor para armazenar a posição do primeiro elemento não nulo dentro de cada coluna armazenado no primeiro vetor. Para uma matriz A de ordem $\underline{n}$ e que apresente $\underline{k}$ elementos não nulos utilizaremos uma área de 2k + n posições para armazenar a matriz. Tomemos, por exemplo, uma matriz A de ordem 200 e com os seguintes elementos não nulos: a_{21}, a_{41}, a_{52}, a_{13}, a_{33}, a_{24} e a_{45}. A ma-

triz será armazenada por colunas como se pode facilmente verificar. Os vetores serão da seguinte forma:

VALOR = (a_{21}, a_{41}, a_{52}, a_{13}, a_{33}, a_{24}, a_{45})

LINHA = (2, 4, 5, 1, 3, 2, 4)

INDCOL= (1, 3, 4, 6, 7)

Os vetores VALOR e LINHA possuem o mesmo número de elementos. A cada elemento de VALOR corresponde um valor respectivo em LINHA,que informa o valor associado da linha. No vetor INDCOL teremos em cada um dos seus n componentes o valor do primeiro elemento não nulo em termos de posição no vetor VALOR. Isto é, o elemento a_{21} está na linha 2 e na coluna 1 da matriz original. Seu valor está armazenado em VALOR (1) e é o primeiro elemento não nulo da coluna 1. No vetor LINHA temos em LINHA (1) o valor da linha correspondente ao elemento armazenado em VALOR (1). Temos em INDCOL (1) a posição do primeiro elemento não nulo da primeira coluna armazenado em VALOR. Outro modo de visualizarmos o processo é o seguinte: para recuperar o elemento a_{33} entramos em INDCOL (3) e obtemos o valor 4. De posse deste valor entramos em LINHA (4) e obtemos o valor 1. Este é o valor da linha do primeiro elemento não nulo da coluna 3. Deste modo, LINHA (4) ou um de seus elementos seguintes,que sejam anteriores ao primeiro elemento não nulo da coluna 4, contem o valor desejado 3 para a linha, desde que a_{33} seja $\neq 0$. Neste caso LINHA (5) = 3, e então em VALOR (5) obtemos o valor de a_{33} procurado. Para o exemplo temos as seguintes medidas:

a) Total de elementos = 40.000 (40.000 posições).

b) Elementos não nulos = 7

c) Área total compactada = 2 * 7 + 5 = 19 posições.

d) Percentagem da área total utilizada = 0.0475% e

e) Índice de esparsidade = 99.9825%.

Analisemos, agora, outra situação onde, para uma matriz A de ordem 200, temos 99 elementos não nulos por linha. Agora temos as seguintes medidas:

a) Total de elementos = 40.000 (40.000 posições).

b) Elementos não nulos = 99 * 200 = 19.800 posições.

c) Área total compactada = 2 * 99 * 200 + 200 = 39.800 posições.

d) Percentagem da área total utilizada = 99.50%.

e) Índice de esparsidade = 50.50%.

Para este método um índice de esparsidade maior ou igual a 50.50% é aceitável. Para índices menores do que este limite seria melhor utilizarmos algum outro processo.

5.2.2.3 ARMAZENAMENTO COMPRIMIDO COM DOIS VETORES

Este método caracteriza-se por associar a cada elemento não nulo de uma matriz esparsa um número inteiro único designado por <u>t</u>, calculado do seguinte modo:

$$t(i, j) = i + (j - 1)\, n$$

onde $a_{ij} \neq 0$

Para armazenar a matriz usa-se dois vetores: um chamado VALOR onde estão os valores dos elementos não nulos, e outro chamado IND, onde estão os valores de <u>t</u> calculados como exposto acima. O vetor IND contém os valores de <u>t</u> correspondentes aos valores de a_{ij} que estão armazenados em VALOR (J), onde

J = 1, 2, ..., k, e k é o número de elementos não nulos da matriz.

Analisemos o seguinte exemplo onde uma matriz A de ordem 200 apresenta os seguintes elementos não nulos: a_{21}, a_{41}, a_{52}, a_{13}, a_{33}, a_{24} e a_{45}. Os vetores mencionados são os seguintes:

VALOR = (a_{21}, a_{41}, a_{52}, a_{13}, a_{33}, a_{24}, a_{45})

IND = (2, 4, 205, 401, 403, 602, 804)

Suponhamos que haja necessidade de recuperar o quinto elemento do vetor VALOR. Inicialmente entramos em IND (5) e obtemos o valor 403. Qual o valor de j e de i para definirmos o elemento a_{ij} procurado? O valor de j pode ser calculado como sendo o menor inteiro maior ou igual a t/n. O valor de i pode ser calculado por i = t (i, j) - (j - 1)n. O inteiro maior ou igual a 403/200 é 3, e neste caso j = 3. O valor de i é (403 - (3-1) 200) = 3. Concluímos então, que o elemento armazenado em VALOR (5) é o correspondente ao elemento a_{33} da matriz original.

Para o exemplo em estudo temos as seguintes medidas:

a) Total de elementos = 40.000 (40.000 posições).

b) Elementos não nulos = 7.

c) Área total compactada = 14 posições.

d) Percentagem da área total utilizada = 0.0350%.

e) Índice de esparsidade = 99.9825%.

Consideremos agora outra situação de uma matriz A que possua ordem 200 e apresente 100 elementos não nulos por linha. Passamos a ter as seguintes medidas:

a) Total de elementos = 40.000 (40.000 posições).

b) Elementos não nulos = 100 * 200 = 20.000.

c) Área total compactada = 2 * 20.000 = 40.000 posições.

d) Percentagem da área total utilizada = 100%.

e) Índice de esparsidade = 50.0%.

No método em estudo podemos utilizar matrizes com esparsidade maior ou igual a 50%.

Voltamos a lembrar que para matrizes com características particulares outros procedimentos poderiam ser buscados. No nosso caso, como estamos fazendo uma análise geral e não sabemos, à priori, o tipo de matriz esparsa que vamos tratar, procuramos não levar em conta estas características especiais.

5.2.2.4 ARMAZENAMENTO COM UM VETOR COMPACTADO

Neste método os elementos não nulos de uma matriz esparsa são armazenados em um vetor chamado, por exemplo, de VALOR, e é usado um outro vetor chamado, por exemplo, de IND, onde estão os subscritos das colunas dos elementos correspondentes. Adicionalmente certas informações de controle são armazenadas em IND também.

Em computadores como o B6700 onde a palavra ou posição de memória tem um comprimento que permita armazenar valores inteiros com mais de dez dígitos, é possível, através de alguns artifícios matemáticos, armazenar dois valores na mesma palavra com um comprimento, cada um, de cinco dígitos. No

B6700 a palavra mede 48 bits ou 6 bytes. Para entender o método basta imaginar que um número será armazenado nos primeiros 24 bits ou na metade, à direita, e que outro número será armazenado nos outros 24 bits ou na metade, à esquerda da mesma palavra do B6700. No caso de B6700, a palavra pode armazenar um valor inteiro de até 12 dígitos valendo, no máximo, 549755813887. Voltando ao método, podemos analisar agora a estrutura do vetor IND, observando o gráfico a seguir:

Palavra	Metade esquerda	Metade direita
(1)	Número de linhas	Número de colunas
(2)	Número total de elementos não nulos armazenados no vetor	
(3)	Número de elementos não nulos na linha 1	Número de elementos não nulos na linha 2
(4)	Número de elementos não nulos na linha 3	....
.	.	.
.	.	.
.	.	.
(I)	.	Número de elemento não nulos na última linha
(I+1)	Número da coluna do primeiro elemento armazenado	Número da coluna do segundo elemento armazenado
(I+2)	Número da coluna do terceiro elemento armazenado	....
.	.	.
.	.	.
(J)	.	Número da coluna do último elemento armazenado

Não podem haver espaços em branco no vetor IND, isto é, se o número de linhas da matriz é ímpar, o índice da primeira coluna deve aparecer na metade direita da palavra que contém "Número de elementos não nulos da última linha" na

metade esquerda.

O vetor IND ocupa uma área igual a (4 + (número de linhas) + (número de elementos não nulos) + 1) / 2, arredondando o resultado, se fracionário, para baixo, para o inteiro mais próximo.

Consideremos o exemplo a seguir, onde uma matriz A de dimensões (4,3) cujos os elementos não nulos são os seguintes: a_{11}, a_{13}, a_{21}, a_{22}, a_{42}. Os vetores terão a seguinte estrutura:

VALOR = (1., 2., 2., 3., 1.)

IND = (400003, 5, 200002, 000001, 100003, 100002, 200000)

Para compreender a formação do vetor IND basta seguir as instruções dadas para sua formação anterior. Estamos considerando, no exemplo, que cada meia palavra armazena um número de cinco dígitos.

Analisemos um segundo exemplo, onde uma matriz A de ordem 200 possui três elementos não nulos por linha. Para este caso temos as seguintes medidas:

a) Total de elementos = 40.000 (40.000 posições).

b) Elementos não nulos = 3 * 200 = 600.

c) Área total compactada = 600 + (4 + 200 + 600 + 1) / 2 = 1.002 posições.

d) Percentagem da área total usada = 2.505%.

e) Índice de esparsidade = 98.5%.

Para efeito de comparação consideremos uma matriz A de ordem 200 que apresente 132 elementos não nulos por linha. As medidas para este caso serão:

a) Total de elementos = 40.000 (40.000 posições).

b) Elementos não nulos = 132 * 200 = 26.400.

c) Área total compactada = 26.400 + (4 + 200 + 26.400 + 1) / 2 = 39.702.

d) Percentagem da área total usada = 99.255%.

e) Índice de esparsidade = 34%.

Podemos observar que este método se aplica para matrizes com índices de esparsidades maiores ou iguais a 34% com vantagens de redução de memória.

Este último método foi proposto por |23| e os demais por |1|. Tendo em vista que pretendemos tratar matrizes esparsas de caráter geral e que, à priori, não sabemos nada sobre elas, nem seu índice de esparsidade e nem suas dimensões, achamos que este último método seja o mais apropriado para a construção de nossos algorítmos para tratamento de matrizes esparsas no B6700. Observe o leitor que para o método escolhido podemos trabalhar com matrizes de até 99.999 linhas por 99.999 colunas e com índice de esparsidade maior ou igual a 34%, o que nos dá uma ampla faixa de funcionamento. Passaremos, em seguinda, a analisar as principais operações a serem realizadas com matrizes esparsas nesta pesquisa e também a filosofia usada para a construção dos algorítmos a serem apresentados e discutidos posteriormente.

5.3 OPERAÇÕES COM MATRIZES ESPARSAS

A construção dos algorítmos discutidos posteriormente, no ítem 5.4, teve como diretriz básica o método

selecionado na seção anterior e que foi aquele exposto no ítem 5.2.2.4, onde eram utilizados dois vetores: um para armazenamento dos valores não nulos e um outro para armazenamento de índices e informações de controle. Para analisar as operações com matrizes esparsas e o seu impacto nos algoritmos, devemos lembrar o que foi apresentado no capítulo 2 sobre memória virtual e os problemas de paginação que devemos evitar. As considerações feitas naquele capítulo sobre o modo como o compilador FORTRAN armazena matrizes, também deve ser considerado para que possamos atingir um padrão de eficiência aceitável. Um aspecto importante em nosso trabalho é que podemos empregar o método exposto no ítem 5.2.24, armazenando a matriz esparsa por linhas em um vetor e deste modo os algoritmos poderão continuar a ser escritos por linhas, sem nenhuma alteração. O único trabalho adicional será o de realizarmos as operações de armazenamento e de recuperação dos elementos não nulos através de rotinas especialmente preparadas para isto, e não deixar que o próprio compilador construa seus próprios procedimentos para tal.

Desta maneira, os algorítmos projetados visam utilizar todos os elementos não nulos de uma linha e das seguintes que estejam armazenados no mesmo segmento carregado na memória, e somente será requisitado um novo segmento quando o anterior já estiver esgotado em termos de utilidade para o processamento. Desta maneira, fica evidenciada a preocupação em construir algorítmos que procurem um balanceamento entre o uso de memória principal tempo de entrada e saída, redução das operações de paginação e também a busca de soluções que não introduzam grande complexidade nos algorítmos, visto

que poderão ser usados por usuários de áreas das mais diversas e que não necessariamente são experts em computação.

A seguir, passamos a descrever as operações que julgamos ser as mais importantes no manuseio de matrizes esparsas, sem, no entanto, reconhecer que para aplicações especiais talvez outras devessem ser construídas e analisadas. Gostaríamos de lembrar também que este trabalho não tem por objetivo resolver sistemas de equações lineares, mas, apenas, apresentar de um modo claro e eficiente, formas de armazenar, recuperar e tratar matrizes esparsas.

5.3.1 LEITURA DE UMA MATRIZ

Julgamos ser fundamental a existência de uma rotina que permita ao usuário fornecer ao sistema desenvolvido, os valores não nulos e todas as informações de controle necessárias para a montagem do vetor com os valores não nulos e para o vetor de controle. As informações sobre os valores não nulos e as de controle podem ser fornecidas via cartões perfurados ou também por disco, fita ou similar, desde que os registros apresentem um lay-out compatível com aquele descrito neste ítem. A subrotina LEIA permite a entrada de uma matriz esparsa, com os elementos fornecidos por linha em ordem ascendente de colunas. Após o término de cada linha, um sentinela deve ser fornecido e que pode ser um qualquer valor. Para definição de cada elemento não nulo dentro de uma certa linha, é necessário fornecer o seu valor e em seguida um índice que defina a coluna a que ele pertence dentro daquela li

nha. Após a colocação de um sentinela para indicar fim de linha, usamos um índice da forma 90000 + I, onde I é o número da linha. Para indicar o fim da matriz usamos um sentinela que pode ser qualquer número seguido de um índice 99999. Os elementos são fornecidos ao sistema no formato 4(E15.8,I5), sendo definidos em pares com o valor do elemento não nulo seguido de um índice de coluna. Antes de se partir para a definição dos elementos não nulos deve-se fornecer ao sistema o número de linhas da matriz, o número de colunas, e o número de elementos não nulos da matriz. Estas informações devem ser fornecidas no formato (5, I5, I5) e servem para efetuar um controle sobre os elementos definidos, a seguir, em termos de linhas, colunas de número total de elementos da matriz.

Esta subrotina recebe uma matriz na forma descrita e armazena as informações lidas em um vetor para os valores não nulos e outro vetor para as informações de controle e índices. Ela foi construída para permitir a manipulação de matrizes de até 9999 linhas e 9999 colunas, e permite que se defina até 65.535 elementos não nulos. Se for necessário trabalhar com um número superior de elementos não nulos, basta realizar algumas pequenas alterações de tal modo que se utilize mais de um vetor de controle e mais de um vetor de armazenamento de valores, porém este trabalho não procurou solucionar problemas de tal ordem. Para que se tenha uma idéia das potencialidades do método escolhido, apresentaremos, agora, algumas considerações sobre o porte das matrizes que podemos tratar diretamente sem a introdução de quaisquer modificações nos algorítmos. Inicialmente gostaríamos de mencionar que a limitação de até no máxima 65.535 elementos não nulos

se deve ao fato do compilador FORTRAN que utilizamos no B6700 do NCE/UFRJ permite que qualquer "array" tenha no máximo aquele comprimento, e utilizamos, por comodidade na implementação dos algorítmos, apenas um "array" daquele tamanho para armazenamento dos valores não nulos. Daí se pode compreender que para armazenar um número superior de elementos não nulos basta utilizar um ou mais vetores de armazenamento e aplicar o mesmo critério para armazenar as informações de controle em mais de um vetor.

Para podermos avaliar melhor o método usado, lembremos que foi definido anteriormente o índice de esparsidade como sendo a percentagem de elementos nulos em relação ao número total de elementos da matriz. Vamos utilizar, também, um índice que pode ser visto como o complemento do índice de esparsidade, que doravante chamaremos de densidade ou índice de ocupação. Entre o índice de esparsidade I1 e o índice de ocupação I2 existe a seguinte relação: I1 + I2 = 1. Por isso dissemos que são complementares. Tomemos o caso de uma matriz quadrada de ordem 255, que é o maior tamanho de matriz que o compilador FORTRAN usado pode tratar diretamente. Tal matriz possuí 65.025 elementos no total. Façamos variar o número de elementos não nulos e procuremos verificar como se comportam os índices de esparsidade e de ocupação. Outro fator que pode ser de muito interesse é a área total usada para armazenar os elementos não nulos e as informações de controle.

Nº elementos não nulos	Índice de Esparsidade	Índice de Ocupação	Área Ocupada
1.000	98,47	1,53	1.630
5.000	92,32	7,68	7.630
10.000	84,62	15,37	15.130
15.000	76,94	23,06	22.630
20.000	69,25	30,75	30.130
30.000	53,87	46,13	45.130
40.000	38,49	61,51	60.130
43.263	33,47	66,53	65.024

Podemos observar que para matrizes de ordem menor ou igual a 255, que poderiam ser armazenadas integralmente na memória, visto que o compilador trata matrizes até aquela dimensão, o método se adequada perfeitamente para índices de esparsidade maiores ou iguais a aproximadamente 33,47%.

Para aumentar as dimensões das matrizes a serem tratadas, devemos inicialmente frisar que o usuário não poderá utilizar o compilador diretamente, pois fora do limite especificado para "arrays" a responsabilidade é do usuário. Por opção de construção estamos armazenando os elementos não nulos em um vetor apenas, e por isso estamos limitados em, no máximo, 65.535 elementos por vetor. Para observarmos o impacto de tal decisão de construção do sistema, procuremos analisar o que acontece com o índice de esparsidade e o índice de ocupação à medida que tratamos matrizes de um porte maior. O quadro a seguir mostra as limitações e potencialidades do método quando usado para o tratamento de um porte maior e que não poderiam ser alocadas diretamente na memória em um único

"array" em FORTRAN.

Ordem da Matriz	Número de elementos	Índice de Esparsidade - Mínimo	Índice de Ocupação - Máximo
260	67.600	3,06	96,94
300	90.000	27,2	72,80
350	122.500	46,60	53,40
400	160.000	59,10	40,90
450	202.500	67,64	32,35
500	250.000	73,80	26,20
1.000	1.000.000	93,50	6,50
2.000	4.000.000	98,40	1,60
4.000	16.000.000	99,60	0,40
5.000	25.000.000	99,74	0,26
9.999	99.980.001	99,94	0,06

Apenas para dar uma ilustração adicional, consideremos o caso de tratarmos uma matriz de ordem 9.999. Para o armazenamento de tal matriz com um índice de esparsidade de 99,94% utilizaríamos a seguinte quantidade de memória principal:

65.535 + (4 + 9999 + 65.535 + 1) / 2 = 65.535 + 37.769 = 103.304 posições.

Voltamos a dizer que podemos tratar matrizes maiores ainda do que estas mencionadas no quadro, porém acreditamos que da forma que o sistema foi calibrado, já podemos atender a uma vasta gama de problemas práticos.

5.3.2 IMPRESSÃO DE UMA MATRIZ ESPARSA

Para facilitar a conferência e a preparação de saídas que possam ser aproveitadas diretamente em relatórios técnicos, o sistema oferece uma subrotina para impressão da matriz em estudo em um formato que facilite também a leitura. Somente os elementos não nulos são impressos, linha por linha, e com cinco elementos por linha. Cada elemento é seguido pelo seu índice de coluna. Cada linha é precedida pelo cabeçalho "Linha número I" e, em cada início de página, aparece uma cabeçalho escolhido pelo usuário. O tamanho da página que foi ajustado para 50 linhas pode ser facilmente adaptado para qualquer outro valor.

Para a impressão da matriz esparsa em estudo, o vetor com informações de controle tem de ser consultado e por isso não é impresso também. O nome desta subrotina é IMPRIM.

5.3.3 IMPRESSÃO DO VETOR DE CONTROLE

Esta subrotina auxiliar foi implementada para permitir a fácil conferência dos valores impressos pela subrotina descrita em 5.3.2. O vetor é impresso seqüencialmente, com mensagens que facilitam a compreensão e para sua montagem é usado o esquema apresentado em 5.2.2.4. O nome desta subrotina é MOSTRA.

5.3.4 GRAVAÇÃO DA MATRIZ EM MEMÓRIA AUXILIAR

Para permitir que a matriz esparsa possa ser usada em vários processamentos não consecutivos e também para facilitar e melhorar as operações de entrada, o sistema oferece uma subrotina que permite a gravação da matriz esparsa em disco, fita ou similar em formato comprimido. Tanto a matriz esparsa como seu vetor de controle são armazenados em um arquivo de número N definido pelo usuário. Esta subrotina é muito apropriada para o caso em que desejamos manipular várias matrizes esparsas num mesmo processamento e não há memória principal suficiente para conter todas elas, mesmo estando em formato comprimido. O nome desta rotina é GRAVE.

5.3.5 LEITURA DE MATRIZ DA MEMÓRIA AUXILIAR

Para possibilitar a recuperação de matrizes esparsas armazenadas em forma comprimida a partir da memória auxiliar existe a subrotina PEGUE que recupera uma matriz gravada no arquivo N definido e criado anteriormente. O vetor de controle da matriz é recuperado igualmente. O uso conjugado desta subrotina com aquela definida em 5.3.4 pode permitir a resolução de muitos casos em um único processamento apesar das restrições de memória principal.

5.3.6 CÁLCULO DA TRANSPOSTA DA MATRIZ

Uma operação muito comum é a definição da

transposta de uma dada matriz. Para uma certa matriz dada como entrada para a rotina, é definido como saída; sua matriz transposta e seu respectivo vetor de controle. Em caso de erros mais comuns a rotina fornece mensagens apropriadas para correção por parte do usuário. O nome desta rotina é TRANSP. A matriz original continua intacta após a operação.

5.3.7 MOVER UMA MATRIZ ESPARSA PARA OUTRA

Esta rotina recebe como entrada uma certa matriz esparsa e o seu vetor de controle e move o seu conteúdo para outra área definida pelo usuário. A operação também é realizada para o vetor de controle. O nome desta rotina é MOVA. Após o término da operação a matriz original continua intacta.

5.3.8 COPIAR UMA LINHA DA MATRIZ

Para copiar uma qualquer linha de uma matriz esparsa o sistema oferece a subrotina COPIAL. O resultado é armazenado em um vetor definido pelo usuário, em forma descompactada, isto é, incluindo todos os elementos nulos. Testes são feitos para verificar a compatibilidade dos tamanhos das áreas manipuladas e, em caso de erro, mensagens apropriadas são enviadas para o usuário proceder as correções necessárias.

5.3.9 COPIAR UMA COLUNA DA MATRIZ

Para copiar uma qualquer coluna da matriz esparsa definida na entrada, o sistema oferece a subrotina COPIAC. O resultado é armazenado em um vetor definido pelo usuário, em forma descompactada, isto é, incluindo todos os elementos nulos. Em caso de erro de tamanho de área alocada, mensagens apropriadas são enviadas ao usuário para as devidas correções.

5.3.10 TROCAR O CONTEÚDO DE DUAS LINHAS

Para trocar o conteúdo de duas linhas especificadas de uma dada matriz esparsa, o sistema oferece a subrotina TROCAL. O resultado é armazenado em uma outra área definida pelo usuário, montando também seu respectivo vetor de controle. A matriz original continua intacta.

5.3.11 TROCAR O CONTEÚDO DE DUAS COLUNAS

Para trocar o conteúdo de duas colunas especificadas de uma dada matriz esparsa, o sistema oferece a subrotina TROCAC. O resultado é armazenado em uma outra área definida pelo usuário, e o seu vetor de controle respectivo também é criado. A matriz original continua intacta.

5.3.12 PERMUTAR AS LINHAS DE UMA MATRIZ

Para permitir a permutação das linhas de uma dada matriz esparsa, o sistema oferece uma subrotina chamada PERML, que efetua as trocas necessárias seguindo as informações fornecidas pelo usuário no vetor de permutações que é fornecido como parâmetro de entrada. O resultado da operação é armazenado em uma área definida pelo usuário para conter a nova matriz e outra para o seu vetor de controle. A matriz original continua intacta. Testes são feitos para detetar possíveis erros de dimensionamento de áreas ou no vetor de permutações fornecido pelo usuário, enviando mensagens apropriadas em cada caso.

5.3.13 PERMUTAR AS COLUNAS DE UMA MATRIZ

Para possibilitar a permutação das colunas de uma dada matriz esparsa o sistema oferece uma subrotina chamada PERMC. As operações são efetuadas com base em um vetor contendo informações dadas pelo usuário para as trocas necessárias. O resultado será armazenado em uma área definida pelo usuário, área esta para conter a nova matriz e outra para o seu respectivo vetor de controle. A matriz original continua intacta.

5.3.14 SOMA DE DUAS MATRIZES ESPARSAS

Para permitir a soma de duas matrizes espar-

sas quaisquer, o sistema oferece a subrotina SOME que armazena o resultado em uma área definida pelo usuário para conter a nova matriz e outra para o seu respectivo vetor de controle. Testes são realizados para testar a validade das áreas dimensionadas pelo usuário e,em caso de erro, mensagens apropriadas são enviadas para as correções necessárias. As matrizes originais continuam intactas.

5.3.15 SOMAR UMA LINHA I1 A UMA LINHA I2 n VEZES

Para se conseguir somar uma linha I1 n vezes a uma linha I2 de uma certa matriz esparsa, o sistema oferece a subrotina SOMEL. A soma é acumulada na linha I2. Esta rotina armazena a matriz resultante em uma área definida pelo usuário e outra para o seu respectivo vetor de controle. A matriz original continua intacta após o término das operações.

5.3.16 SOMAR UMA COLUNA C1 A UMA COLUNA C2 n VEZES

Para se conseguir somar uma coluna C1 n vezes a uma outra coluna C2 o sistema oferece a subrotina SOMEC. A soma é acumulada na coluna C2. Para conter os resultados o usuário deve dimensionar uma área para a nova matriz e outra para o seu respectivo vetor. A matriz original continua intacta após o término das operações. Testes são efetuados e, em caso de erros, mensagens apropriadas são enviadas para efetuar as correções necessárias.

5.3.17 MULTIPLICAR UMA LINHA DA MATRIZ POR UM ESCALAR

Para permitir a multiplicação de uma qualquer linha de uma matriz esparsa por um escalar definido pelo usuário o sistema oferece uma subrotina chamada MULTL, que armazena os resultados em uma outra área definida pelo usuário para conter a nova matriz e o seu vetor de controle. A matriz original continua intacta. Testes são feitos e, em caso de erros mais comuns, mensagens apropriadas são enviadas ao usuário para correções devidas.

5.3.18 MULTIPLICAR UMA COLUNA DA MATRIZ POR UM ESCALAR

A multiplicação de uma qualquer coluna de uma matriz esparsa pode ser feita através da subrotina chamada MULTC, que armazena os resultados em uma área definida pelo usuário capaz de conter a nova matriz e o seu respectivo vetor de controle. A matriz original continua intacta. Testes são realizados para garantir a validade das áreas dimensionadas e, em caso de erros mais comuns, mensagens apropriadas são enviadas para que o usuário faça as correções apropriadas.

5.3.19 MULTIPLICAR TODOS OS ELEMENTOS DE UMA MATRIZ POR UM ESCALAR

A multiplicação de todos os elementos de uma matriz esparsa por um escalar definido pelo usuário pode ser feita através do uso de uma subrotina chamada MULTES, que ar-

mazena os resultados em uma área definida pelo usuário, capaz de conter a nova matriz e o seu respectivo vetor de controle. A matriz original continua intacta. Testes são efetuados para garantir a validade das áreas dimensionadas e, em caso de erros mais comuns, mensagens apropriadas são enviadas ao usuário para que ele possa efetuar as correções necessárias.

5.3.20 MULTIPLICAÇÃO DE DUAS MATRIZES ESPARSAS

Para efetuar a multiplicação de duas matrizes esparsas, o sistema oferece a subrotina MULTIP que multiplica uma matriz A pela matriz transposta de B armazenando o resultado em C, criando também o vetor de controle respectivo. A matriz deve estar armazenada em colunas para efetuar a multiplicação. As matrizes originais permanecem intactas. Testes são feitos para garantir a validade das áreas dimensionadas pelo usuário e, em caso de erro, mensagens apropriadas são fornecidas para possibilitar as correções necessárias.

5.3.21 ARMAZENAMENTO DE INFORMAÇÕES NO VETOR DE CONTROLE

Para que se possa armazenar informações no vetor de controle de uma matriz esparsa, o sistema oferece a subrotina COMPAC que armazena as informações descritas em 5.2.2.4, usando parte da palavra correspondente para fins de redução do uso de memória. Esta subrotina é usada para a maioria das outras subrotinas mencionadas anteriormente.

5.3.22 RECUPERAÇÃO DE INFORMAÇÕES DO VETOR DE CONTROLE

Uma vez definido e criado o vetor de controle de uma matriz esparsa qualquer, basta usar a função RETIRA para recuperar as informações definidas pela subrotina do ítem 5.3.21. Esta função é utilizada em diversas subrotinas mencionadas anteriormente.

5.3.23 INVERSÃO DE UMA PERMUTAÇÃO

Mencionamos anteriormente a possibilidade de realizar permutações de linhas e de colunas de uma matriz esparsa a partir de um vetor de permutações fornecido pelo usuário. O sistema oferece também uma subrotina para realizar a inversão de uma certa permutação, que é chamada de INVPER. O resultado é armazenado em outro vetor e o vetor original é deixado intacto após o término das operações.

5.3.24 INVERSÃO DE UMA MATRIZ ESPARSA

A inversão de uma matriz esparsa pode ser conseguida através da utilização da subrotina INVERT que recebe uma matriz esparsa qualquer e seu vetor de controle como parâmetros de entrada e devolve a matriz resultante e o seu vetor de controle. A inversão não é uma operação trivial e podemos recair em um caso onde a matriz de entrada não é inversível ou então a matriz de saída não é esparsa ou ainda não apresenta as características de esparsidade desejadas de tal

modo que seja compatível com os recursos de software e de hardware disponíveis nas instalações do usuário. Como já discutimos anteriormente no capítulo, os métodos diretos reduzem a esparsidade ou, em outras palavras, aumentam a densidade de matrizes esparsas e não são recomendáveis para este tipo de operação. Os métodos interativos por sua vez podem ter problemas sérios de convergência muito lenta ou de não convergirem realmente para alguns casos analisados pelos autores citados no capítulo quatro. Para efetuarmos a inversão de matrizes utilizamos o método proposto em |35| que procura manter a matriz inversa tão esparsa quanto possível. O método é chamado pelos autores de "Inversão de matrizes pela forma do produto da inversa (PFI)". Estudos mais profundos sobre este assunto podem ser encontrados em |35| e em |1| que analisam, inclusive, casos particulares de matrizes esparsas, o que não é o nosso objetivo nesta pesquisa.

Após esta breve descrição das operações selecionadas para implementação, por considerarmos que são as mais importantes para o tema central deste trabalho, passamos em seguida a apresentar os algorítmos desenvolvidos em FORTRAN para o B6700 do NCE/UFRJ. Para não tornar o texto muito extenso e também para ressaltar os aspectos interessantes dos mesmos, incluímos na discussão apenas aquelas partes que nos pareceram mais relevantes.

Para maiores informações ou obtenção de cópias dos mesmos basta entrar em contacto com os autores deste trabalho.

5.4 OPÇÕES DE OTIMIZAÇÃO DO FORTRAN

Após as definições feitas no capítulo anterior, passamos a elaborar algorítmos capazes de realizar as operações julgadas como as mais comuns e de maior importância na solução de problemas que apareçam em matrizes esparsas. É evidente que na implementação dos algorítmos, procuramos titar partido das características específicas do B6700 buscando uma otimização sempre que possível, com as opções oferecidas pelo próprio compilador e as características de "hardware" do B6700. Isto não significa, no entanto, que procuramos utilizar comandos do FORTRAN que sejam particulares, apenas para tal equipamento. Sempre que possível, procuramos utilizar instruções básicas da linguagem, facilitando, deste modo, futuras tarefas de conversão para outro tipo de equipamento. Como principais características usadas para otimizar os algorítmos projetados e implementados no B6700 podemos citar as seguintes:

5.4.1 OPÇÃO DE OTMIZAÇÃO DO COMPILADOR

O compilador FORTRAN do B6700, como em muitos outros sistemas conhecidos, oferece aos usuários uma opção de compilação representada por OPT, que pode ser usada para alterar o código objeto gerado para um programa através da alteração do modo como o compilador trata instruções individuais e também grupos de instruções fornecidas no programa fonte do usuário. A forma de se pedir esta otimização é a seguinte:

$SET OPT n   ou então, $ SET OPT = n

onde o valor do argumento (n) pode assumir os valores 0,-1 e 1. Procuramos utilizar, sempre que possível, o valor de (n) igual a 1 que nos conduz a uma otimização muito mais poderosa. Esta opção deve ser fornecida ao início de um programa ou subprograma. No caso de subprogramas que possuam um "label" como um parâmetro formal, ou qualquer outro subprograma que referencie tal subprograma como descrito, não pode ser otimizado.

Em casos onde o compilador não consegue realizar a otimização pedida, será gerado um erro de sintaxe. Como a opção pode ser definida para cada função, subprograma ou subrotina separadamente, nada impede sua utilização nos módulos adequados mesmo que alguns não sejam apropriados para tal otimização. Para maiores informações sobre o uso desta opção de compilação, basta recorrer a |13|.

5.4.2 SEGMENTAÇÃO DE MATRIZES

Como já foi discutido no capítulo dois, o sistema B6700 realiza segmentação automática de matrizes acima de 4095 palavras sem a intervenção do usuário. Se nos reportarmos ao tópico 5.3, veremos que em todos os casos estamos usando o esquema escolhido em 5.2.2.4 onde, basicamente, temos um vetor para armazenar os valores não nulos da matriz esparsa e um segundo vetor para guardar as informações de controle da própria matriz. Em nenhum caso foi necessário utilizar vetores maiores do que 65.535 palavras, que constituem o máximo permitido nesta implementação do compilador FORTRAN disponível.

O compilador FORTRAN permite o uso de uma opção de compilação chamada LONG que permite ao usuário alterar a maneira como o FORTRAN trata as matrizes na memória. A forma para usar esta opção é a seguinte:

$SET LONG

As matrizes declaradas em partes do programa fonte onde a opção LONG seja válida não serão segmentadas, não importando seu tamanho desde que o limite anteriormente especificado não seja ultrapassado. Esta opção pode ser desligada em qualquer parte de um módulo da seguinte maneira:

$RESET LONG

Em virtude de nossos algorítmos não utilizarem memória em excesso para a configuração que possuímos, procurando usar esta opção em todos os módulos.

5.4.3 SEGMENTAÇÃO DO CÓDIGO GERADO

Uma das saídas fornecidas pelo compilador FORTRAN do B6700 é o programa objeto que é chamado de "code file". O programa objeto é logicamente segmentado por módulos, tais como subprogramas. O código de cada módulo começa em um segmento físico do disco e preenche tantos segmentos quantos necessários, dentro dos limites do sistema. Um módulo extremamente grande pode ter necessidade de mais de um segmento e, neste caso, o compilador automaticamente fará tal divisão. No momento que o tamanho do código alcance a 8.192 palavras ou 30*A (onde A é o valor numérico da AREASIZE do code file),

dependendo de qual dos dois valores for menor, tal segmentação ocorrerá. O código para o programa objeto é então contido em dois segmentos de programa. Esta segmentação pode ocorrer mais de uma vez, dependendo do tamanho do módulo em questão.

No caso em que o programa principal for compilado separadamente das subrotinas e subprogramas, tal segmentação só ocorrerá para o programa principal.

O parâmetro AREASIZE mencionado anteriormente indica o número de registros lógicos em uma área de um arquivo em disco. Deve ser sempre um número inteiro e para um melhor aproveitamento de espaço, deve ser divisível pelo BLOCKSIZE que é o tamanho do bloco. O valor assumido por omissão é o número mais próximo a 1000 que seja divisível pelo BLOCKSIZE. Seu valor máximo é 1 048 575.

Normalmente, todo arquivo deveria ter determinados atributos definidos pelo próprio usuário que tem muito mais condição de fazer uma boa escolha do que deixar o sistema fazê-lo pois, certamente, sua eficiência não será tão boa já que não conta com todas as informações necessárias para especificá-los. Tais parâmetros são os seguintes:

a) MAXRECSIZE

Este atributo especifica o tamanho máximo do registro lógico, levando-se em conta os atributos INTMODE e UNITS.

O atributo INTMODE especifica o modo interno de armazenamento dos caracteres e pode assumir os seguintes valores:

0 - palavra binária de 48 bits

2 - HEX (4 bits)

3 - BCL (6 bits)

4 - EBCDIC (8 bits)

5 - ASCII (8 bits)

O atributo UNITS indica o tipo de representação que pode ser em caracteres ou em palavras que terão os atributos MAXRECSIZE e BLOCKSIZE. O atributo UNITS pode assumir os seguintes valores:

0 - WORDS (palavras de 48 bits)

1 - CHARACTERS (no código definido por INTMODE)

O tamanho máximo permitido para o MAXRECSIZE é 65 535. Quando não é especificado ou possua valor zero, um valor será atribuído pelo sistema que dependerá do tipo de periférico em uso e também do valor do BLOCKSIZE. Se ambos forem omitidos ou nulos, serão escolhidos em função do tipo de periférico em uso, como se pode ver a seguir:

PERIFÉRICO	MAXRECSXIZE	BLOCKSIZE
DISCO	30	30
TERMINAL	12	12
IMPRESSORA	22	22
LEITORA CARTÕES	14	14
PERFURADORA	14	14
FITA MAGNÉTICA	10	10

Os valores mencionados estão todos em palavras de 48 bits.

b) BLOCKSIZE

Este parâmetro indica o tamanho do registro físico. Se o parâmetro UNITS for igual a 1 o BLOCKSIZE é dado em unidades do atributo INTMODE. Se o valor de UNITS for igual a zero então teremos o valor de BLOCKSIZE em palavras de 48 bits. O valor máximo de BLOCKSIZE é 65 535 palavras. Quando não é especificado, as considerações feitas para o MAXRECSIZE também aqui se aplicam. Uma boa norma é definir o valor do BLOCKSIZE como múltiplo de 30 palavras para que não percamos espaço devido ao alinhamento em segmentos no disco. Um cuidado deve ser tomado pois o crescimento do BLOCKSIZE implica em uma alocação maior de memória principal.

c) AREASIZE

Este parâmetro indica o número de registros lógicos em uma área de um arquivo em disco. Deve ser sempre um número inteiro e para melhor utilização da área em disco, deve ser divisível pelo BLOCKSIZE. O valor que o sistema atribuí por omissão deste parâmetro é igual a um número mais próximo possível a 1000 que seja divisível pelo BLOCKSIZE. Seu valor máximo é 1 048 575.

d) AREAS

O parâmetro AREAS indica o número máximo de áreas que um arquivo em disco pode ter. No caso do FORTRAN, quando seu valor é omitido o sistema assume AREAS=20. Seu va

lor máximo permitido é 1000. Uma boa norma para aplicações estáticas onde o número de registros não varia com o tempo é manter o valor de AREAS entre 25 e 30. Para arquivos dinâmicos que não atendam as características do tipo anterior, devemos ter AREAS entre 30 e 150 desde que em um valor de AREASIZE não seja superior à 3600 segmentos. Cada segmento mencionado em todo o presente capítulo assumimos o valor de 30 palavras.

Em resumo, para um certo valor de MAXRECSIZE conhecido e igual a M e para UNITS=0, que é assumido pelo sistema, o valor do BLOCKSIZE pode ser calculado lembrando-se que deve ser múltiplo do MAXRECSIZE e do tamanho do fator de blocos do sistema que é igual a 30 palavras. Isto pode ser dito do seguinte modo:
(M * N) MOD 30 = 0, onde N é um número inteiro e de preferência menor do que 300 para evitar um excessivo consumo de memória principal. Em seguida, podemos calcular o valor da AREASIZE por AREASIZE = (BLOCKSIZE * W) / MAXRECSIZE, onde W deve assumir um valor menor ou igual ao número de registros do arquivo. O valor de W deve ser tal que o valor calculado para o parâmetro AREAS fique entre 25 e 30 para arquivos estáticos e entre 30 e 150 para arquivos dinâmicos. Passamos, em seguida, para o cálculo de AREAS, onde AREAS = (Número de registros) / AREASIZE e, no caso de número fracionário como resposta, devemos arredondar para cima para chegarmos a um número inteiro.

Estas considerações sobre tais parâmetros foram apresentadas neste trabalho pois devem afetar considera-

velmente a execução em termos de sobrecarga para o sistema, que poderia ser minimizada se o usuário tivesse o cuidado de preparar uma especificação adequada para o arquivo que contém a saída da compilação e, deste modo, ajudar ao sistema nas tarefas de segmentação e de futuras operações de entrada e saída. O que já vimos anteriormente pode ser um problema formidável, principalmente em nosso caso que tratamos de matrizes de grandes dimensões e de características bem particulares de esparsidade. Uma escolha adequada pode ser uma excelente contribuição para chegarmos a um padrão de otimização durante o processamento.

5.4.4 SEGMENTAÇÃO CONTROLADA PELO USUÁRIO

Os procedimentos apresentados em 5.4.3 podem ser alterados através da opção de compilação chamada SEGMENTATION que permite segmentar um programa de forma diversa. A forma de usá-la é a seguinte:

$SET SEGMENTATION = n

onde (n) é o tamanho do segmento em palavras. Em vez de um número podemos usar a palavra MAX, que será interpretada como igual a 65 535 palavras. No caso de haver disponibilidade de memória, como é o nosso caso, esta é uma boa opção e deve ser considerada.

Outra opção associada que força a segmentação de forma diversa da que foi exposta anteriormente é aquela que pode ser conseguida através do uso da opção NEWSEGMENT.

O aparecimento da opção $SET NEWSEGMENT em meio ao programa fonte, força que as instruções fornecidas em seguida sejam armazenadas em um novo segmento do arquivo de código. Esta opção só pode ser usada no programa principal. Se durante a execução uma referência for feita a um segmento não residente, uma operação de "overlay" deverá ser realizada. Por isto a escolha do local onde tal opção deve ser colocada no programa deve ser criteriosamente escolhida. Por exemplo, tal opção não deveria ser colocada dentro de um ciclo de um comando DO pois isto pode gerar operações desnecessárias para o sistema operacional.

5.4.5 COMPILAÇÕES SEPARADAS

Ao invés de escrevermos um programa e todas as subrotinas necessárias para realizar as funções definidas na seção anterior em 5.3 e compilarmos tudo junto o que poderia gerar um código muito extenso, podemos usar a opção SEPARATE que permite a geração de um arquivo de código objeto para cada módulo do pacote para manipulação de matrizes esparsas que vamos implementar. Tais arquivos de códigos isolados podem ser unidos a outros do próprio usuário ou de outros através do uso de BINDER. Esta opção pode ser ligada ou desligada através de $SET SEPARATE ou $RESET SEPARATE. Tal opção é válida para todos os módulos do programa fonte e não pode ser especificada para alguns módulos específicos.

5.4.6 USO DO 'VECTOR MODE'

O Vector Mode é uma opção de "hardware" de alguns sistemas B6700 que podem permitir um aumento significante na eficiência do tratamento de matrizes e de ciclos como os formados pelos comandos DO ou IF/GOTO. Esta opção de hardware só pode ser usada dentro de determinadas situações que procuraremos resumir aqui, porém quando tais restrições são atendidas, esta opção é de muita valia. Uma discussão mais profunda pode ser vista em |13|. As condições básicas para uso do VECTOR MODE são as seguintes:

a) O compilador FORTRAN tenha sido compilado com as opções OPTLSS1 e VECTORMODEISALLOWED, sendo a primeira desligada e a segunda ligada.

b) A opção OPT deve ser ligada com um valor maior do que zero para o módulo respectivo e que tal otimização seja possível.

c) A opção VECTORMODE tenha sido ligada para o módulo em questão.

d) Além destas restrições gerais, a própria estrutura do módulo deve atender a determinadas características mandatárias e que são as seguintes:

d.1) O módulo deve possuir um ciclo ("loop") formado por um comando DO ou formado pelo uso conjugado de IF/GOTO.

d.2) Só é possível entrar no ciclo por meio do seu primeiro comando e nunca em um ponto qualquer, intermediário, dentro do ciclo.

d.3) A variável de controle do ciclo não pode assumir valores para o incremento superior a 3.

d.4) Os valores inicial, final e o incremento devem ser inteiros e não podem ser alterados dentro do próprio ciclo para a variável de controle do ciclo.

d.5) A variável de controle não pode estar em uma área de um comando COMMON.

d.6) Dentro do ciclo não podem haver instruções dos tipos relacionados a seguir:

- CALL
- Comandos de entrada e saída
- PAUSE
- ZIP
- Comandos auxiliares para tratamento de arquivos
- Referências a funções
- Referências a intrínsicos, inclusive exponenciação.

d.7) Caso existam matrizes ou variáveis em COMMON que sejam alteradas dentro de um ciclo, ou mesmo que sejam apenas referenciadas dentro do ciclo usando subscritos que variem a cada iteração, tais operações devem estar limitadas a um máximo de três. Isto não significa que um máximo de três matrizes pode ser usado em um ciclo.

d.8) Todas as matrizes devem ser declaradas com a opção LONG para evitar que sejam segmentadas.

d.9) Somente os ciclos mais internos são apropriados para tratamento por VECTORMODE.

Mais importante do que todas estas considerações feitas é a de que, para que a opção VECTORMODE seja possível, é necessário que o processamento esteja equipado com o "hardware" próprio para tratamento desta opção, sem o qual o programa terminará de modo anormal. Um tratamento mais profundo deste assunto pode ser encontrado em |13| onde maiores exemplos são apresentados. Um estudo mais profundo sobre os tópicos abordados em todo o ítem 5.4 pode ser encontrado em |34| e em |37| onde sugestões específicas são apresentadas pelos autores para buscar um padrão de eficiência maior em aplicações de tipos variados.

5.5 ALGORÍTMOS IMPLEMENTADOS

Com base nas definições e intenções apresentadas em 5.3 onde falamos de operações com matriz esparsas pudemos implementar os seguintes algorítmos:

1 - Entrada de dados de forma conversacional aproveitando as características de "time sharing" que o B6700 procura oferecer aos seus usuários. Este módulo permite também que o usuário obtenha uma listagem do arquivo criado em disco e que automaticamente se torna disponível para tratamento pelas demais rotinas do sistema de tratamento de matrizes esparsas. O arquivo criado contém a própria matriz de forma descompactada e que pode ser lido pela rotina descrita a seguir.

2 - Leitura de um arquivo (em cartões, disco, fita, ...) que contenha a matriz esparsa de forma descompactada, com todos

os seus elementos não nulos e esta é armazenada ao final em dois vetores, um para os valores propriamente ditos e outro para as informações de controle da matriz.

3 - Impressão da matriz esparsa armazenada em dois vetores conforme descrito em (2). Para a impressão do relatório esta rotina permite, ainda, que um título seja fornecido para o cabeçalho identificando a matriz ou o processamento realizado com a matriz.

4 - Impressão do vetor que contém as informações de controle. Esta rotina se mostra bem útil para auxiliar na depuração e conferências necessárias por parte do usuário.

5 - Gravação da matriz de forma compactada em um arquivo de modo a permitir sua futura utilização.

6 - Leitura de uma matriz esparsa na forma compactada a partir de um arquivo criado pela rotina descrita em (5).

7 - Cálculo da transposta de uma matriz esparsa dada sem destruí-la. Uma nova matriz é criada com seu vetor de controle próprio.

8 - Movimentação de uma matriz esparsa de uma área para outra definida pelo usuário. Uma nova matriz igual à anterior é criada sem destruir a matriz original.

9 - Cópia de uma qualquer linha de uma matriz esparsa escolhida e o resultado é armazenado em um vetor descompactado, isto é, com todos os seus elementos nulos e não nulos.

10 - Cópia de uma qualquer coluna de uma matriz esparsa escolhida e o resultado é fornecido da mesma forma que a descrita em (9).

11 - Trocar o conteúdo de duas linhas quaisquer de uma matriz esparsa escolhida pelo usuário criando uma nova matriz sem no entanto destruir a matriz original.

12 - Trocar duas colunas quaisquer de uma matriz esparsa escolhida pelo usuário criando uma nova matriz sem destruir a matriz original.

13 - Permutar as linhas de uma matriz esparsa escolhida pelo usuário, criando uma nova matriz sem detruir a matriz original. As permutações a serem realizadas são definidas pelo usuário através de um vetor de permutações fornecido como entrada.

14 - Permutar as colunas de uma matriz esparsa escolhida pelo usuário e uma nova matriz será criada em outra área, sem destruir a matriz original. As permutações são definidas pelo usuário através de um vetor de permutações.

15 - Somar uma linha de uma matriz esparsa a outra, ambas escolhidas pelo usuário tantas vezes quanto for pedido, criando uma nova matriz sem destruir a matriz original.

16 - Somar uma coluna à outra de uma matriz esparsa escolhida pelo usuário um determinado número de vezes e armazenando a nova matriz em outra área, mantendo assim a matriz original intacta para futuras utilizações.

17 - Compactação dos subscritos das colunas no vetor de controle da matriz esparsa utilizando um esquema de manipulação parcial de palavras como descrito em 5.2.

18 - Descompactação das informações do vetor de controle de uma matriz esparsa escolhida pelo usuário e fornecendo a informação desejada sem destruir o vetor de controle.

19 - Inversão de uma permutação armazenada em um vetor fornecido pelo usuário na entrada e um novo vetor é criado sem destruir o anterior.

20 - Cálculo da soma de duas matrizes esparsas escolhidas pelo usuário e o resultado é armazenado em uma terceira matriz definida também pelo usuário.

21 - Multiplicação de uma determinada linha de uma matriz por uma constante definida pelo usuário e a matriz resultante será armazenada em uma área definida de modo a não destruir a matriz original.

22 - Multiplicação de uma determinada coluna de uma matriz esparsa por uma constante criando uma nova matriz sem destruir aquela que foi fornecida como parâmetro de entrada para a rotina.

23 - Multiplicação de uma matriz esparsa por um escalar qualquer definido pelo usuário criando uma nova matriz sem destruir a matriz original.

24 - Multiplicação de duas matrizes esparsas criando uma terceira matriz e mantendo as duas matrizes originais intactas.

25 - Subtração de uma matriz esparsa de outra matriz esparsa armazenando o resultado em uma terceira matriz definida pelo usuário. As matrizes de entrada são conservadas intactas para futuras utilizações.

26 - Inversão de uma matriz esparsa armazenando o resultado em uma outra matriz definida pelo usuário e conservando a matriz de entrada intacta após o término das operações.

Os programas desenvolvidos e implementados no B6700 não foram introduzidos no corpo do texto, pois tornaria o mesmo extenso demais, e se houver interesse do leitor, os mesmos podem ser conseguidos com os autores desta pesquisa que pertencem à COPPE/UFRJ.

Os algorítmos propostos em 5.3 foram desenvolvidos e testados usando-se para isto matrizes de diversos tamanhos e de diferentes graus de esparsidade e nos pareceram ser adequados para as aplicações que utilizamos. Acreditamos que, na medida em que tais rotinas sejam utilizadas por uma gama de usuários de áreas diversas, as mesmas possam ser depuradas em situações que talvez não tivessem sido percebidas pelos autores e também acreditamos que possam ser adaptadas a problemas particulares para tirar vantagem das características específicas de certos tipos de matrizes esparsas. Obviamente, adaptações poderão ser feitas de modo a explorar melhor as características do equipamento à disposição do usuário. Lembramos ao leitor que procuramos usar as opções básicas do FORTRAN para facilitar a conversão e implementação dos algorítmos citados em outras instalações que não apenas a que uti

lizamos, o NCE/UFRJ.

Finalmente, gostaríamos de ressaltar que não foi nosso objetivo construir um sistema que cobrisse todas as operações possíveis com matrizes esparsas, mas apenas mostrar que são possíveis e acreditamos que muitas outras podem ser criadas pelo leitor.

6. CONCLUSÕES E RECOMENDAÇÕES PARA A IMPLANTAÇÃO DE CÓDIGOS EM PROGRAMAÇÃO LINEAR

A hipótese implícita nesta pesquisa é a de que é possível a criação de algorítmos particulares para matrizes esparsas que permitam um nível de eficiência (em termos de tempo de processamento, de tempo de entrada e saída e de utilização de memória) que seja superior aos índices que os métodos desenvolvidos para matrizes de alta densidade apresentam. Para compreensão dos métodos tradicionais para manipulação de matrizes não esparsas desenvolvemos os capítulos quatro e cinco. Para identificação dos problemas que as aplicações tradicionais normalmente apresentam em termos de processamento, desenvolvemos o capítulo dois.

Finalmente, procuramos caracterizar melhor os conceitos relacionados à matrizes esparsas no capítulo cinco. Algumas alternativas de armazenamento foram apresentadas e discutidas. Uma das alternativas foi escolhida e diversos algorítmos foram projetados e desenvolvidos à luz dos conceitos apresentados no capítulo dois.

Em seguida, implementamos os algorítmos em FORTRAN do B6700 do NCE/UFRJ e realizamos diversos testes com os métodos tradicionais usando rotinas desenvolvidas e de largo uso para manipulação de matrizes de alta densidade como aquelas do IMLS, do SSP e outras apresentadas em livros de Cálculo Numérico e nossas rotinas mostraram ser mais eficientes do que as demais no tratamento de matrizes esparsas, princi-

palmente quando o índice de esparsidade é elevado. À medida que as matrizes vão se tornando mais densas, os algorítmos vão piorando em termos de performance, mais ainda, são comparáveis com os algorítmos tradicionais. Para compreendermos a qualidade dos algorítmos desenvolvidos, diversos testes tiveram que ser realizados e com base naqueles testes pudemos tirar um conjunto de conclusões que passamos a enumerar a seguir, e sempre que possível, apresentando as justificativas que nos parecem mais adequadas. Acreditamos que muitos outros testes podem e devem ser realizados e que estamos apenas começando a pesquisar e buscar soluções na prática para tal assunto que desperta tanto interesse de áreas das mais diversas. Como principais conclusões do presente trabalho podemos citar as seguintes:

1 - O simples fato de um computador possuir memória virtual não libera o usuário de ter que exercer um controle no desenvolvimento de suas aplicações para evitar situações altamente indesejáveis, onde o tempo dedicado às operações de entrada e saída, executadas pelo sistema operacional, se torna excessivo. Em aplicações com matrizes de pequenas dimensões isto pode passar até desapercebido por parte do usuário, mas no caso de aplicações onde grandes matrizes, e no nosso caso esparsas, são tratadas, este problema se torna muito mais sério. Em áreas como Engenharia Civil, Engenharia Elétrica e outras este problema é até muito comum. Quando ao invés de um computador de grande porte e com memória virtual, o usuário dispõe em sua instalação de um computador com uma reduzida memória, o problema de gerenciar o uso de memória se torna mandatário.

Neste trabalho procuramos, sempre que possível, dar facilidades ao usuário para manipular matrizes esparsas de grandes dimensões, sem, no entanto, utilizar áreas de memória de forma desnecessária, ou seja, procuramos minimizar o uso deste recurso tão importante em um sistema de processamento de dados.

2 - O conhecimento de como o compilador empregado para desenvolver uma aplicação do tipo que tratamos neste trabalho é fundamental para a criação de algorítmos mais eficientes que aqueles que são usualmente apresentados em livros de cálculo numérico para tratamento de matrizes densas. Se o compilador escolhido para o projeto, o FORTRAN em nosso caso, armazena uma matriz de um determinada forma e todos os algorítmos não procuram tirar proveito desta informação, isto, certamente, conduzirá a uma sub-utilização dos recursos computacionais existentes. Para o nosso estudo concluímos que para usar o armazenamento por colunas, que é regularmente usado pelo compilador FORTRAN, os algorítmos deveriam ser reescritos para tratar as variáveis indexadas por colunas e não por linhas como são usualmente apresentadas nos livros textos. Para fazermos tais afirmações algumas análises conceituais foram realizadas no capítulo dois e testes respectivos foram realizados no B6700 para comprovar tais afirmações. Em nosso sistema optamos por outra forma de armazenamento que foi por linha. Os algorítmos assim, estão como são usualmente apresentados nos livros que tratam do assunto de matrizes. Apenas procuramos aproveitar as características de esparsidade das matrizes e construímos algorítmos que manipulassem apenas os elementos não nulos das matrizes em questão. Desta maneira, o armazenamento e a recu-

peração de elementos da matriz esparsa são feitos por rotinas que fazem parte do sistema e não deixamos que o FORTRAN crie ou utilize seus próprios procedimentos para executar tais tarefas.

3 - O armazenamento de matrizes sob a forma vetorial se mostrou adequado para o tratamento das matrizes esparsas, sem aumentar demasiadamente a complexidade dos algorítmos implementados no B6700. Notamos, no entanto, que ao armazenarmos uma matriz esparsa por linha, como foi o nosso caso, a recuperação de alguns elementos por coluna se torna uma operação ineficiente quando comparada com a recuperação por linha, que é a forma empregada no sistema. Nos casos em que julgamos necessário utilizar recuperação por colunas, optamos por trabalhar com a transposta da matriz para então recairmos naquele caso onde nossos algorítmos são mais eficientes.

4 - A decisão de criar uma subrotina para tratar cada uma das operações definidas em 5.3 para as matrizes mostrou-se muito razoável, pois fornece uma flexibilidade maior ao usuário que pode chamar em seu programa apenas aquelas subrotinas que lhe interessam para um processamento específico.

5 - A operação de inversão de uma matriz esparsa não é uma operação trivial e geralmente caímos no caso da matriz inversa não apresentar um índice de esparsidade compatível com as restrições de memória principal existente, o que certamente obriga o usuário a ter um conhecimento muito mais profundo sobre o tratamento de matrizes inversas como discutido em |1| e em |35|. Como já vimos, os métodos diretos para definição da ma

triz inversa que são conhecidos conduzem a erros de arredondamento que podem disvirtuar totalmente as soluções encontradas. As operações de "scaling" das linhas e também das colunas têm sido contestado por alguns autores |1| por julgarem eles que tais operações não conduzem, em todos os casos, a uma situação de equilíbrio. Os métodos iterativos por sua vez, podem ter sérios problemas de convergência e não podemos garantir nada à priori. Existem estudos que procuram criar condições para contornar este problema como descrito em |19|. Acreditamos que para solucionar problemas de programação linear, talvez seja mais adequado utilizar os métodos conhecidos para solução de sistemas e introduzindo as devidas adaptações para o caso de matrizes esparsas, em termos de armazenamento, como apresentamos neste trabalho, do que tentar inverter uma matriz esparsa e esbarrar num problema de encontrarmos uma matriz inversa que apresente uma alta densidade e não podermos resolver o problema.

6 - As subrotinas desenvolvidas procuram tratar matrizes esparsas de quaisquer tipos, pois não conhecemos as características que as mesmas podem apresentar à priori. É de se esperar que se as devidas adaptações forem introduzidas para tirar proveito das características específicas de determinados tipos de matrizes esparsas, que os algorítmos possam ter sua performance melhorada para aquele caso considerado. Por exemplo, o tratamento de matrizes simétricas, diagonais ou de um dos tipos apresentados em 3.5 e cujo tratamento é analisado em |1|.

7 - A utilização de uma opção de compilação que permita que o vetor que contém a matriz esparsa esteja integralmente na memória principal, sem ser segmentado, gerando operações desnecessárias de paginação, mostrou ser de muita utilidade e contribuiu sensivelmente para reduzir o "elapsed time" dos testes. No nosso caso esta opção foi a opção LONG descrita em 5.4.2.

8 - A conjugação da opção LONG e da opção OPT de otimização introduzida pelo próprio compilador trouxe reduções do tempo de processamento e de entrada e saída, em alguns casos de mais de 10% dos tempos anteriormente obtidos sem o uso de tais opções.

9 - Não pudemos utilizar em todos os algorítmos desenvolvidos a opção VECTORMODDE em virtude da quantidade de condições impostas para seu uso, porém nos casos em que pudemos conjugar o uso desta opção com as descritas em (8) obtivemos uma redução de tempo de processamento de entrada e saída e também de "elapsed time" que variaram numa faixa de 5% a 20%. Não foi possível chegar a valores muito precisos em função da dificuldade de comparação de dois algorítmos num sistema como o B6700 do NCE/UFRJ, onde existe uma carga variável de trabalho ao longo do dia, carga esta que varia também em função do dia da semana como pudemos observar. Como nunca pudemos rodar dois algorítmos sozinhos, dentro da máquina, não podemos chegar a valores precisos em termos da otimização obtida em termos de tempo. Em termos de memória, fica um pouco mais fácil analisar, na média, o gasto que os algorítmos apresentaram, pois o próprio sistema calcula e fornece tais estatísticas e as ne-

cessidades dos programas não variam de um processamento para outro. De uma forma geral, quando passamos a empregar as opções oferecidas pelo próprio compilador para otimização, os tempos foram reduzidos de forma sensível, isto podemos afirmar.

10 - O conhecimento de como definir adequadamente os arquivos de saída do compilador com base nas observações apresentadas em 5.4.3. permitiram que o código gerado fosse de maior eficiência, do que deixando o próprio compilador definir as características dos arquivos por "default", como a maioria dos usuários deixam acontecer em nossa instalação.

11 - O tamanho da palavra do B6700 é de tal ordem que não tivemos problemas em implementar a filosofia do vetor de controle apresentada em 5.2.2.4, utilizando manipulação parcial de palavra com possibilidade de tratamento de matrizes de ordem até 9 999, e, certamente, poderíamos aumentar tal ordem de grandeza, porém, acreditamos já haver chegado a um limite considerável, onde a maioria dos problemas práticos costumam chegar. Pudemos implementar esta filosofia sem ter a necessidade de utilizar ALGOL ou ASSEMBLER do B6700 para tais operações, usando tão somente instruções básicas do FORTRAN.

Estas conclusões apresentadas foram aquelas que mais nos chamaram a atenção entre as várias facetas do trabalho desenvolvido e implementado em FORTRAN do equipamento B6700 do NCE/UFRJ.

Não esperamos, no entanto, haver esgotado o assunto, mas em verdade apenas começado a analisar e verificar

os problemas práticos existentes na grande área de Programaçaõ Linear. Tratamos de um pequeno tópico sobre as matrizes esparsas. Muitas outras idéias foram sugeridas em |1| e que não tivemos a oportunidade de colocar em prática. Esta foi uma primeira aproximação do problema; não que tenhamos a pretensão de ser originais ou de dar uma grande contribuição à ciência, mas de estudar um assunto interessante e de real aplicação prática a curto prazo por nossos colegas, que se defrontam com este problema de tratar matrizes esparsas em suas aplicações em áreas diversas e de dar como produto final não só as conclusões do estudo, mas também um pacote para realizar aquelas operações que nos parecem as mais comuns no manuseio de matrizes esparsas de uma forma geral. Pedimos, também, que toda e qualquer omissão encontrada nas análises efetuadas e nos algorítmos implementados possam ser trazidas ao nosso conhecimento para que as adaptações necessárias possam ser realizadas e novamente divulgadas para os interessados neste trabalho e seus resultados.

Finalmente, gostaríamos de destacar quais foram as principais contribuições deste trabalho de pesquisa e como podem ser utilizadas pela comunidade.

O primeiro ponto a ressaltar é que colegas de diversas áreas do conhecimento esbarram no tratamento de matrizes esparsas quando tentam resolver problemas que se reduzem à solução de um sistema de equações lineares. Por isso acreditamos que os resultados aqui obtidos podem interessar as seguintes áreas: Engenharia Civil, Engenharia de Sistemas, em problemas de Otimização, Matemática Aplicada, Engenharia Elé-

trica, Métodos Quantitativos aplicados à Administração, e,possivelmente, outras que não conseguimos identificar.

O segundo ponto importante é que aquelas pessoas nem sempre possuem conhecimentos profundos de computação e estaõ muito mais preocupadas em solucionar seus problemas do que projetar algorítmos sofisticados apropriados para matrizes esparsas. Observamos mesmo que tais pessoas geralmente conhecem a linguagem FORTRAN ou similar como o BASIC, como conseqüência de terem participados de cursos daqule tipo em suas universidades de origem.

O terceiro ponto a destacar é que aquela população identificada se localiza parcialmente em Universidades espalhadas pelo país e também em empresas que com baixa freqüência possuem equipamentos de grande porte. Geralmente os equipamentos daquelas instalações possuem o compilador FORTRAN ou o BASIC e igualmente dispõem de pouca memória principal.

O quarto ponto relevante para esta pesquisa é que o Governo Brasileiro, ao criar a reserva de mercado para os fabricantes nacionais de mini-computadores, permitiu que muitas empresas pequenas e médias adquirissem aqueles equipamentos. Em parte daquelas empresas, certamento haverá necessidade de resolver problemas que recaiam no uso de matrizes esparsas, como descrito anteriormente. Naturalmente, os mini-computadores não possuem uma extensa memória principal, e isto pode se constituir numa grande restrição à resolução de problemas de interesse prático.

Com base nos quatro pontos apresentados ante-

riormente, afirmamos que as principais contribuições desta pesquisa foram as seguintes:

a) Discutir alternativas de armazenamento de matrizes esparsas que podem ser implementadas em computadores com pouca memória e mesmo em mini-computadores. Qualquer uma das alternativas discutidas pode ser implementada numa linguagem como o FORTRAN que é amplamente conhecida. As alternativas permitem um uso reduzido de memória principal para armazenar matrizes esparsas, quando comparados com os algorítmos tradicionais. Para fins de verificação empírica, uma das alternativas foi selecionada e os algorítmos foram implementados em FORTRAN no equipamento da NCE/UFRJ.

b) O problema de paginação que ocorre quando os algorítmos são programados em FORTRAN, devido à forma como o compilador armazena matrizes (por colunas), foi resolvido ao se armazenar a matriz esparsa por linha, em forma vetorial, com apenas um índice ou subscrito. Deste modo, os algorítmos escritos, geralmente por linha, não precisaram ser modificados.

c) A implementação de um conjunto de subrotinas, escritas em FORTRAN, para o tratamento de matrizes esparsas utilizando um volume reduzido de memória principal e reduzindo a necessidade de paginação, constituí um produto que certamento pode ser usado, de modo relativamente simples, por parte da população descrita anteriormente.

Acreditamos, deste modo, ter produzido um trabalho que pode ser absorvido sem maiores dificuldades por parte da comunidade acadêmica e empresarial e, para isto, nos colocamos, mais uma vez, à disposição para dar o suporte neces-

sário para que tal absorção realmente ocorra.

7. BIBLIOGRAFIA

1. TEWARSON, Reginald P. Sparse Matrices, New York, Academic Press, 1973. (Mathematics in Science and Engineering Series, 99).

2. MADNICK, Stuart and DONOVAN, John. Operating Systems, New York, McGraw-Hill, 1974. (Computer Science Series).

3. BOEHM, Barry W. Software and its impact: a quantitative assessment. Datamation, Los Angeles, Technical Publishing, 19(5):48-59, May 1973.

4. DENNIS, P. J. Virtual Memory. ACM Computing Surveys, New York, Association for Computing Machinery, 2(3):153-90, Sep. 1970.

5. GUIMARÃES, C.C. Princípios de Sistemas Operacionais, Rio de Janeiro, Campus, 1980.

6. BURROUGHS 6700. Information processing systems. Detroit, May 1972. (Reference Manual, 1058633).

7. BURROUGHS B7000/B6000. System software operational guide. Detroit, May 1977. V.2.

8. BURROUGHS B7700. Systems concepts training material. Detroit, Nov. 1973. (Student Material, 1072550).

9. GOTLIEB, C.C. Data types and structures. Englewood Cliffs, Prentice-Hall, 1978.

10. KNUTH, D.E. The art of computer programming. Reading, MA, Addison-Wesley, 1968.

11. MOROZOWSKI, M.F. Matrizes esparsas em redes de potência; técnicas de operações. Rio de Janeiro, COPPE/UFRJ, 1973. Tese (Mestrado).

12. DUBRULLE, A.A. The design of matrix algorithms for FORTRAN and virtual storage. New York, Nov. 1979 (IBM Manual G320-3396).

13. BURROUGHS B6700/B7700. FORTRAN reference manual. Sep. 1974. (5000458).

14. THE COMTRE Corp. and Sayers, Anthony P. ed. Operating Systems survey. Princeton, Auerbach, May 1972. (AUERBACH Computer Science Series).

15. IBM. Scientific Subroutine package. New York, August 1970. (IBM Manual, GH20-0205-4).

16. SANTOS, V.R. Curso de cálculo numérico. Rio de Janeiro, Livro Técnico, 1972. (Ciência da Computação)

17. CARVALHO, J.P. Introdução à álgebra linear. Rio de Janeiro, Livro Técnico, 1972.

18. DE LA PENHA, G. M. S. and CARAKUSHANSKY, M. S. Álgebra Linear I. Rio de Janeiro, Instituto de Matemática/UFRJ, 1975. Não publicado.

19. DELEEUW, S. Digital computation and numerical methods, New York, McGraw-Hill, 1965.

20. VARAIYA, P.P. Notes on optimization. 3.ed., New York, Van Nostrand Reinhold, 1972.

21. WYLIE, JR., C.R. Advanced engineering mathematics, New York, McGraw-Hill, 1960.

22. SORIANO, H.L. and PRATES, C.L.M. Armazenamento computacional de matrizes em análise estrutural. Rio de Janeiro, COPPE/UFRJ, 1978. (PDD,14/78).

23. MCNAMEE, J.M. A sparse matrix package. ACM Communications, New York, Association for Computing Machinery, 14(4): 265-73, Apr. 1971.

24. MOLER, C.B. Matrix computations with FORTRAN and paging. ACM Communications, New York, Association for Computing Machinery, 15(4):268-70, Apr. 1972.

25. MCNAMEE, J.M. Remark on algorithm 408(F4)-A sparse matrix package. ACM Communications, New York, Association for Computing Machinery, 16(9):578, Sep. 1973.

26. HANSON, R.J. and WISNIEWSKI, J.A. A mathematical programming updating method using modified govens transformations and applied to LP problems. ACM Communications, New York, Association for Computing Machinery, 22(4):245-51, Apr. 1979.

27. REYNOLDS, J.C. REasoning about arrays. ACM Communications, New York, Association for Computing Mahcinery, 22(5): 290-9, May 1979.

28. FISHER, P.C. and PROBERT, R.L. Storage reorganization techniques for matrix computation in a paging environment. ACM Communication, New York, Association

for Computing Machinery, 22(7):405-15, July 1979.

29. TARJAN, R.E. and YAO, A.C. Storing a sparse table. ACM Communications, New York, Association for Computing Machinery, 22(11):606-11, November 1979.

30. SALIM, C.S. and SALIM, H.K. The Sparse system. Rio de Janeiro, PUC - Computer Science Departament/Rio Datacenter, 1970. (Monographs in Computer Science and Computer Applications, 06/70).

31. LAM, B. Methods for solving large sparse indefinite systems of linear equations, Canberra, The Australian National University, Computer Center, July 1976. (Technical Report, 53).

32. SEGETHOVA, J. Elimination procedures for sparse symmetric linear systems of a special structure. Maryland, Computer Science Center, July 1970. (Technical Report, 70-121).

33. MCKELLAR, A.C. and COFFMAN, E.G. Organizing matrices and matrix operations for paged memory systems. ACM Communications, New York, Association for Computing Machinery, 12(3):153-65, March 1969.

34. RANDELL, B. and KUEHNER, C.J. Dynamic storage allocation systems. ACM Communications, New York, Association for Computing Machinery, 11(5):297-306, May 1968.

35. MACULAN, N.F. and BRAGA, L.P.V. Aspectos computacionais em programação linear. Rio de Janeiro, Instituto de

Matemática/UFRJ, agosto de 1980. (Relatório Técnico).

36. VOLLMER, L. Em busca de uma utilização eficiente do pack. Boletim Informativo, Rio de Janeiro, NCE/UFRJ, 6(7): 15-27, Agosto 1978.

37. PRAÇA, E. Como criar arquivos de uma forma eficiente. Boletim Informativo, Rio de Janeiro, NCE/UFRJ, 8(1): 38-43, Janeiro 1980.