O EMPREGO DA FATORAÇÃO LU NA RESOLUÇÃO DE PROBLEMAS LINEARES BLOCO-ANGULARES

por

Francisco de Assis Magalhães Gomes Neto

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E CIÊNCIA DA COMPUTAÇÃO

CAMPINAS - SÃO PAULO
BRASIL

O EMPREGO DA FATORAÇÃO LU NA RESOLUÇÃO DE PROBLEMAS LINEARES BLOCO-ANGULARES

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida por Francisco de Assis Magalhães Gomes Neto e aprovada pela comissão julgadora.

Campinas, 14 de abril de 1989

Profª Drª Ana Friedlander
orientadora

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para o título de mestrado em matemática aplicada.

O EMPREGO DA FATORAÇÃO LU

NA RESOLUÇÃO DE PROBLEMAS LINEARES

BLOCO-ANGULARES

TESE DE MESTRADO

Francisco de Assis Magalhães Gomes Neto

Ana Friedlander - orientadora

abril de 1989

AGRADECIMENTOS

à paciente e condescendente orientadora

ANA

e a seu marido JOSÉ MARIO

aos engenheiros FRANCISCO e MARIA LUIZA

a EDUARDO e MARIA OLINDA

a CLIMENE

e aos colegas de mestrado

na esperança de que todos saibam *mas não divulguem*

os meus motivos para agradecer

SUMÁRIO

INTRODUÇÃO

Um homem deve ler de tudo, um pouco
ou o que puder,
não se lhe exija mais do que tanto,
vista a curteza das vidas
e a prolixidade do mundo
JOSÉ SARAMAGO
(in: O ano da morte de Ricardo Reis)

Problemas de programação linear têm sido resolvidos de forma rotineira desde a introdução do método simplex, feita por G. B. Dantzig[1] na década de quarenta. Particularmente a partir dos anos 60, um sem número de versões diferentes deste método

[1]DANTZIG, G. B. Programming of interdependent activities, II, mathematical model. *Econometrica.* 17 (3-4) : 200-211, july-oct. 1949.

foram implementadas em computadores de diferentes capacidades, visando atender objetivos variados, dentre os quais os mais comuns são: abranger o maior número possível de problemas diferentes, aproveitar ao máximo as características da máquina, minimizar o tempo de processamento, reduzir ao máximo a memória necessária para armazenar os dados, atender a um grande número de usuários diferentes, ser compatível com o maior número de computadores disponíveis, ser barato ou de execução barata e resolver problemas particulares da forma mais eficiente possível.

Dentre todos estes objetivos, o último se justifica principalmente quando se trata de resolver problemas de grandes dimensões, que possuem matrizes enormes. Para muitos desses problemas, a solução somente pode ser alcançada quando se abre mão da abrangência, utilizando programas especializados.

Isto ocorre, especialmente, quando se trabalha com matrizes grandes que apresentam um número considerável de elementos nulos. Geralmente, se estes elementos são tratados convenientemente, é possível reduzir significativamente as quantidades de memória e tempo computacionais exigidos na manipulação das matrizes.

Os problemas aos quais este texto é dedicado — aqueles que possuem estrutura bloco-angular — se enquadram nesta classe, mas vão mais além. Neles, não só as matrizes possuem um grande numero de zeros, como também encontramos os

elementos não nulos dispostos de forma tão particular que é possível tornar o processo de obtenção da solução ótima ainda mais rápido e econômico se adotado um tratamento particular conveniente.

Mas as preocupações do otimizador não podem se restringir apenas aos critérios de tempo e espaço gastos. Quando se pretende resolver problemas de grande porte, há que se tomar em conta os erros inerentes às operações aritméticas feitas por computador. Erros estes que são transferidos aos dados obtidos em cada etapa do processo de otimização e que podem ir se acumulando até assumirem uma proporção tal que inviabilize a obtenção da solução correta do problema.

Tendo isto em mente, R. H. Bartels e G. H. Golub [3] apresentaram um processo de mudança de vértice (no algoritmo simplex) capaz de garantir a confiabilidade dos resultados obtidos sem comprometer em demasia o tempo de execução dos programas computacionais. Tal método, que envolve a decomposição da base em matrizes triangulares, tornou-se, desde então, muito popular dada a comprovada estabilidade numérica que induz.

É tão somente a aplicação da idéia de empregar este tipo de decomposição na resolução daqueles problemas lineares que possuem matrizes com forma bloco-angular que nos vai interessar neste trabalho.

No primeiro capítulo, o problema é apresentado,

juntamente com alguns ramos de atividade em que ele é encontrado. Além disso, são superficialmente descritos os algoritmos específicos atualmente disponíveis para a solução desta classe de problemas.

No segundo capítulo, são expostas em profundidade as caracterísscas do algoritmo ora proposto e de todos os seus passos principais.

Em seguida, no terceiro capítulo, são detalhados os aspectos computacionais do método, incluindo os motivos que levaram à adoção de determinadas técnicas de armazenamento e ao uso de estratégias de pivotamento específicas, dentre outros detalhes de implementação.

O quarto capítulo é dedicado às comparações numéricas dos algoritmos descritos nos capítulos I e II. Nele é exposto e comentado o desempenho relativo de cada método, com ênfase particular para problemas de formulação de rações.

Por fim, no capítulo V, é descrito um processo capaz de acelerar o algoritmo apresentado em II quando se otimiza problemas semelhantes ao de rações.

CAPÍTULO PRIMEIRO

ASPECTOS GERAIS DO PROBLEMA

SEÇÃO 1 FORMULAÇÃO

O nosso objetivo principal é resolver, da forma mais eficiente possível, problemas de programação linear com variáveis canalizadas, que podemos definir genericamente como sendo:

$$\begin{aligned} &\text{minimizar} \quad c^T x \\ &\text{sujeito a} \quad b^L \leq Ax \leq b^U \\ &\qquad\qquad\quad l \leq x \leq u \end{aligned} \tag{I.1.1}$$

Onde:

$x \in \mathbb{R}^n$ representa as variáveis do problema;

$c \in \mathbb{R}^n$ é um vetor de custos associados a x;

$b^L \in \mathbb{R}^m$ são os limites inferiores das restrições;

$b^U \in \mathbb{R}^m$ são os limites superiores das restrições;

$A \in \mathbb{R}^{mxn}$ é a matriz tecnológica do problema;

$l \in \mathbb{R}^n$ fornece os limites inferiores e

$u \in \mathbb{R}^n$ os limites superiores das variáveis.

Entretando, iremos nos dedicar a um conjunto mais restrito de problemas, denominados *bloco-angulares*, cuja característica principal reside no fato da matriz A possuir o seguinte aspecto:

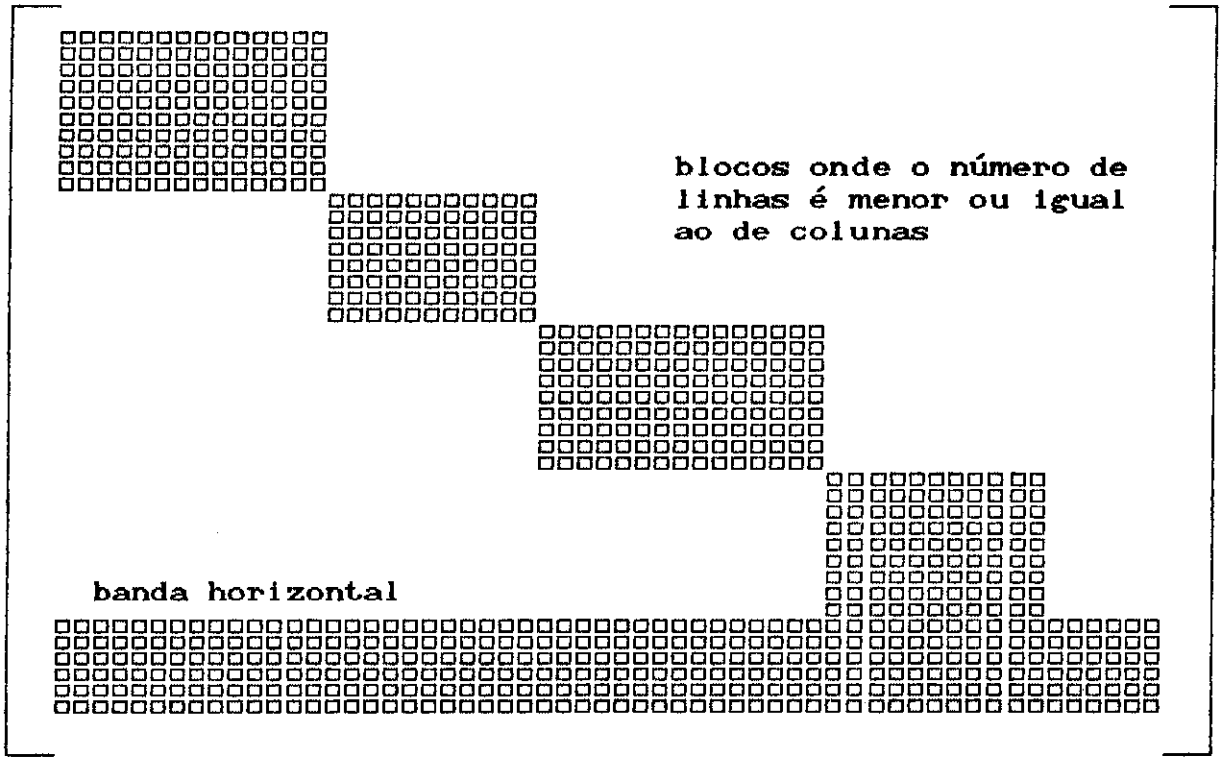


Fig I.1.1 - Uma matriz bloco-angular. Os espaços em branco representam elementos nulos.

O emprego da denominação *bloco-angular* para esta classe de problemas se explica facilmente pela estrutura da matriz acima, na qual se distingue uma série de blocos próximos à diagonal principal, além de uma banda horizontal que, a partir de agora, designaremos pelos termos *restrições de acoplamento* ou *gerais*, ou ainda *restrições de estoque*, em virtude da função a ela associada em grande parte das aplicações reais estudadas.

Antes de passar a descrever qualquer método para resolver os recém apresentados problemas bloco-angulares, parece conveniente comentar ao menos algumas situações reais em que este pode se tornar útil, de forma a fornecer ao leitor uma noção de sua aplicabilidade.

Poderíamos comecar, então, não sem um certo exagero e com fins apenas ilustrativos, dizendo que todo problema de programação linear pode ser reduzido à forma bloco-angular, mesmo que sejamos obrigados, para tanto, a considerar uma matriz tecnológica com apenas um bloco e a retirar deste bloco algumas de suas restrições para compor a banda de acoplamento.

Mas, como dizíamos, embora seja verdade, isto já é um certo exagero, pois, de uma forma geral, dificilmente um algoritmo específico para nosso tipo de problema teria um desempenho médio satisfatório se requisitado para otimizar outros problemas de programação linear com estrutura que não seja intrinsecamente bloco-angular.

Deixemos, portanto, a retórica para trás e passemos à prática, analisando algumas das atividades para as quais

é conveniente a elaboração de um método de solução específico, como o que pretendemos apresentar.

Problemas de alocação de recursos limitados

A aplicação mais freqüente e aquela em que esperamos encontrar os melhores resultados para os algoritmos que estamos estudando é a otimização de problemas compostos de diversas atividades que utilizam recursos comuns, supondo que alguns destes recursos não estão disponíveis em abundância, mas em quantidades limitadas.

Um exemplo clássico é o conhecido *problema da ração* ou *da dieta*, o qual detalharemos em virtude das numerosas referências a ele feitas ao longo do texto.

O problema da ração

Trata-se de produzir diversas rações diferentes, cada uma delas definida através de um sistema na forma:

$$b_i^L \leq A_i x_i \leq b_i^U$$

$$l_i \leq x_i \leq u_i \qquad \text{(I.2.1)}$$

Na expressão I.2.1 acima, i indica tratar-se da i-ésima ração, x_i é o vetor dos ingredientes (milho, soja, por exemplo) utilizados nesta ração, l_i e u_i representam o consumo mínimo e máximo exigido dos ingredientes, os vetores b_i^L e b_i^U fornecem os teores nutricionais (que podem ser de proteínas, vitaminas, cálcio, etc) mínimos e máximos que a ração i deve possuir e as colunas da matriz A_i contêm a composição nutricional de cada ingrediente.

Outra característica muito importante deste problema, e que deve ser levada em conta ainda, é que as rações possuem muitos ingredientes em comum (muitas delas podem conter milho, por exemplo). Por este motivo, não é difícil encontrar, também, uma disponibilidade limitada para alguns destes ingredientes.

Resumindo, dizemos que nosso problema é composto por

1 Diversos *blocos*, cada um descrito na forma I.2.1, representando as rações; e

2 Algumas limitações de estoque para certos ingredientes, o que constitui uma *banda de acoplamento*, que

relaciona elementos de vários blocos, e torna o problema bloco-angular.

Variações

O mesmo problema pode ter outra forma se considerarmos vários lotes de rações que deverão ser produzidos em épocas diferentes. Assim, as restrições de estoque passam a representar a disponibilidade dos produtos limitados ao longo do tempo. A primeira destas equações, por exemplo, indica a quantidade de milho que pode ser consumida pela rações que serão produzidas no primeiro período, a equação seguinte fornece a disponibilidade de milho acumulada até o segundo período (supondo que seja possível estocar o excedente não consumido no período 1), etc.

Com este modelo pode-se, também, otimizar as compras de ingredientes para as rações, admitindo que seus preços variam sazonalmente.

Por fim, outra aplicação clássica do problema de alocação de recursos advém da distribuição de horas de trabalho em locais diferentes (na produção de artigos diferentes) para operários especializados.

Problemas de transporte

Imaginemos, agora, um problema de transporte na forma padrão:

$$\begin{aligned} \text{minimizar} \quad & \sum_i \sum_j c_{ij} x_{ij} \\ \text{sujeito a} \quad & \sum_j x_{ij} = f_i \\ & \sum_i x_{ij} = d_j \\ & 0 \leq x \leq u \end{aligned} \qquad \text{(I.2.2)}$$

onde:

x_{ij} é a quantidade de mercadoria transportada de i para j;

c_{ij} é o custo de transporte entre i e j;

f_i é a disponibilidade do produto na origem i ($f_i>0$); e

d_j é a demanda no destino j ($d_j>0$).

Observa-se que este problema também pode ser escrito na forma bloco-angular se considerarmos, por exemplo, as restrições de oferta como sendo o acoplamento e dividirmos o resto da matriz em blocos que correspondem às restrições de

demanda em cada destino.

Nos casos em que o número de pontos de origem é muito menor que os de destino (ou muito maior, se invertermos os papéis dos blocos e do acoplamento na matriz tecnológica), esta transformação que estamos propondo pode se tornar convidativa e o problema pode ser solucionado de forma eficiente sem ser necessário o uso de um algoritmo especializado em transporte.

Problemas de fluxo de varias mercadorias ou produtos

Consideremos um sistema de vias de transporte de produtos em forma de rede, com seu grafo associado $G(V,A)$, onde V é o conjunto de vértices e A o conjunto de arestas de G.

Suponhamos, agora, que desejamos transportar diversos tipos de mercadorias entre os vários vértices da rede e que cada aresta a_{ij}, ligando os vértices i e j, possui uma orientação (ou seja, nem sempre as mercadorias podem fluir em ambos os sentidos).

Associemos a cada uma destas arestas um limite superior para o volume dos produtos que nela podem ser transportados, u_{ij}, e um custo de transporte que depende também do tipo de mercadoria transportada, c_{pij}.

Por fim, definamos um vetor b_p que indica a demanda ou oferta do produto p na rede (se $b_{pi}<0$, então há demanda do produto p no nó i, caso contrário, se $b_{pi}>0$ há oferta).

Neste caso, podemos dividir o problema por mercadorias, associando a cada uma um bloco de restrições na forma $Ax_p = b_p$, onde A é a matriz de incidência do grafo e x_p é o vetor que representa o fluxo da mercadoria p em cada um dos arcos da rede. O vetor x tem que satisfazer ainda às canalizações $0 \leq x_p \leq u_p$.

Precisamos, ainda, considerar um conjunto de restrições de acoplamento para aqueles arcos onde mais de uma mercadoria pode passar, donde podemos representar o problema na sua totalidade como sendo:

$$
\begin{aligned}
&\text{minimizar} && \sum_p c_p x_p \\
&\text{sujeito a} && \sum_p x_p = u_o \\
& && A.x_p \leq b_p \qquad p = 1, \ldots, m \\
& && 0 \leq x_p \leq u_p \qquad p = 1, \ldots, m
\end{aligned}
\tag{I.2.3}
$$

Também aqui o emprego de algoritmos específicos para modelos bloco-angulares tem sido sugerido e se mostrado eficiente.

Problemas com matrizes na forma escada

Por fim, vamos abordar aqueles problemas nos quais a matriz tecnológica tem forma *escada*, como a da figura I.2.1 abaixo:

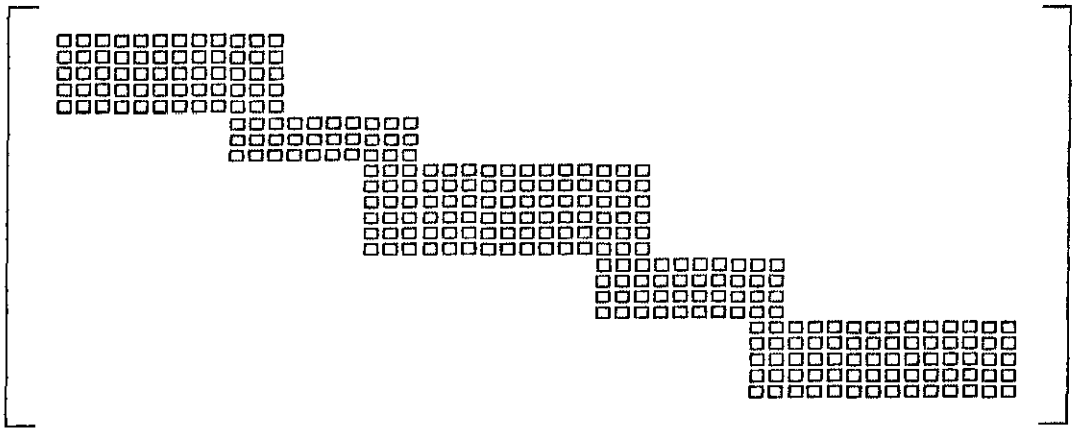


Fig I.2.1 - Uma matriz na forma escada

Um problema deste tipo também pode ser posto na forma bloco-angular se seguirmos os seguintes passos:

1 Primeiro isolamos as colunas que contêm elementos de mais de um bloco na parte direita da matriz:

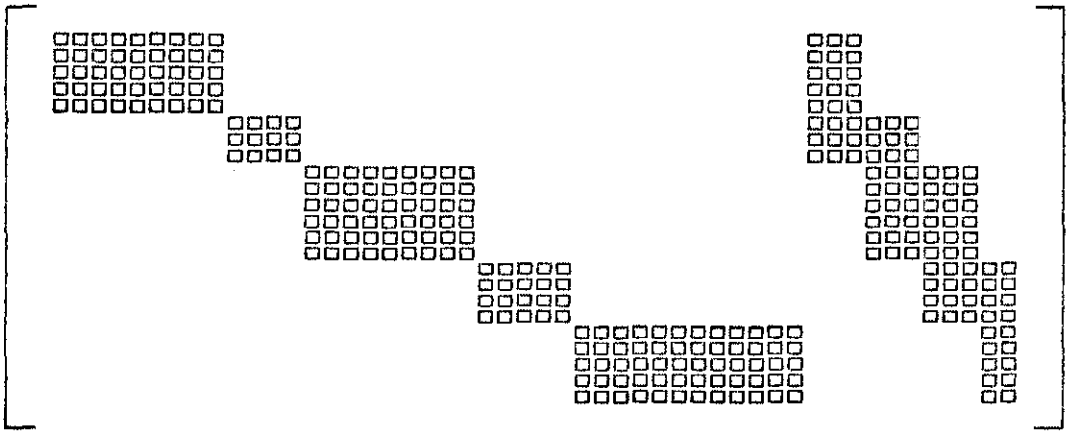


Fig. I.2.2 - A matriz escada com colunas permutadas

2 Depois resolvemos o problema dual, que utiliza A^T, uma matriz bloco-angular.

Entretanto, convém deixar claro que esta transformação deve ser feita com cuidado, não sendo indicada nos casos em que o problema dual contém um número muito grande de colunas ou quando é excessivo o número de colunas que englobam mais de um bloco no problema primal.

Sem tomar por base a seção anterior, que foi muito resumida e superficial, podemos dizer que é considerável o número de problemas que podem ser escritos na forma bloco-angular, motivo pelo qual vários matemáticos de renome já se ocuparam em apresentar algoritmos específicos para sua solução. Alguns destes métodos merecem um destaque maior em virtude de serem considerados bastante eficientes e de serem utilizados largamente. Vamos comentar agora as implementações que consideramos mais interessantes.

Algoritmos de Decomposição

Antes de mais nada, dividamos, outra vez, o nosso problema bloco-angular em duas partes distintas, uma contendo os blocos junto à diagonal, outra comportando as restrições de acoplamento.

Se desejássemos resolver apenas a primeira dessas partes, nosso trabalho se resumiria a otimizar cada bloco em separado, o que é, em geral, muito fácil se comparado ao esforço

necessário para resolver o problema completo (isto ficará bem claro no capítulo correspondente aos resultados computacionais).

Parece ser, então, a primeira vista, proveitoso manter esta divisão se pudermos solucionar separadamente cada um dos subproblemas.

Obviamente, isso não é tão simples, nem é possível conseguir de imediato. Contudo, pode-se lançar mão de algoritmos que, agindo de forma iterativa e operando com o problema dividido, consigam convergir paulatinamente para a solução ótima global.

Observemos como é possível aproveitar essa idéia da decomposição, descrevendo os dois principais algoritmos nela baseados. Naõ sem antes, entretanto, para facilitar o entendimento, re-escrever o modelo formulado na seção 1 deste capítulo na forma:

$$\text{minimizar} \quad c_1^T . x_1^T + c_2^T . x_2^T + \ldots + c_n^T . x_n^T \qquad \text{(I.3.1)}$$

$$\text{sujeito a} \quad A_1 . x_1 = b_1 \qquad \text{(I.3.2)}$$

$$A_2 . x_2 = b_2 \qquad \text{(I.3.3)}$$

$$\vdots$$

$$A_n . x_n = b_n \qquad \text{(I.3.4)}$$

$$D_1 . x_1 + D_2 . x_2 + \ldots + D_n . x_n = f \qquad \text{(I.3.5)}$$

$$x_1 \geq 0 \, , \; x_2 \geq 0 \, , \ldots , \; x_n \geq 0 \qquad \text{(I.3.6)}$$

A idéia de Dantzig e Wolfe

Dantzig e Wolfe[1] foram os primeiros a apresentar um algoritmo de decomposição, no início dos anos 60. Eles partiram da idéia de que, supondo limitada a região factível do poliedro correspondente as restrições dos blocos (equações I.3.2 - I.3.4 e I.3.6)[2], toda e qualquer solução, x, do problema I.3.1 - I.3.6 acima pode ser escrita em função dos vértices, x^1, ..., x^r, deste poliedro, na forma

$$x = \sum_i \lambda_i x^i \qquad \text{(I.3.7)}$$

$$\sum_i \lambda_i = 1 \qquad \text{(I.3.8)}$$

$$\lambda_i \geq 0 \qquad \text{(I.3.9)}$$

Assim, substituindo estes termos na função objetivo I.3.1 e na equação I.3.5, obtém-se

---

[1]DANTZIG, G. B. & WOLFE, P. The decomposition algorithm for linear programming. *Econometrica*. 9 (4):767-778, oct. 1961.

[2]Esta suposição pode ser relaxada, mas evitaremos fazê-lo aqui para facilitar a apresentação do método.

$$\text{minimizar} \quad \sum_i (c.x^i).\lambda_i \qquad \text{(I.3.10)}$$

$$\text{sujeito a} \quad \sum_i (D.x^i).\lambda_i = f \qquad \text{(I.3.11)}$$

$$\sum_i \lambda_i = 1, \ \lambda_i \geq 0 \qquad \text{(I.3.12)}$$

Observando com atenção, nota-se que este problema, que denominaremos *principal*, é absolutamente equivalente ao original, com a vantagem de possuir apenas uma linha a mais que o acoplamento, em comparação ao grande número de linhas do problema I.3.1 - I.3.6.

Mas, por outro lado, ele tem, ao mesmo tempo, tantas colunas quantos vértices contém o poliedro I.3.2 - I.3.4 (em que se considera somente os blocos), valor que geralmente é muito grande.

O que os dois pesquisadores sugerem que façamos é resolver este novo problema, tomando, entretanto, o cuidado de não guardar suas colunas (e tampouco enumerar seus vértices), mas gerá-las quando necessário, ou seja, quando forem tomar parte na base, segundo o seguinte algoritmo:

r Seja π o vetor das variáveis duais correspondentes a equação I.3.11 e α a única variável dual relativa à equação

I.3.12. Os custos relativos, $\tilde{c}_j$, do problema I.3.10 - 1.3.12 podem, então, ser escritos como:

$$\tilde{c}_j = c.x^j - [\pi, \alpha].\begin{bmatrix} D.x^j \\ \hline 1 \end{bmatrix} \qquad (I.3.13)$$

ou, de outra forma,

$$\tilde{c}_j = (c - \pi_1.D).x^j - \alpha \qquad (I.3.14)$$

Usando o critéro de Dantzig para escolher a variável s que entrará na base, temos:

$$\tilde{c}_s = \min_j \tilde{c} = (c - \pi_1.D).x^s - \alpha \qquad (I.3.15)$$

Como os pontos extremos podem ser separados por blocos, resolver I.3.15 é equivalente a encontrar

$$\tilde{c}_s = \sum_i z_i - \alpha \qquad (I.3.16)$$

onde i, índice que representa o bloco, varia entre 1 e n, e z_i é obtido resolvendo-se o problema

$$\text{minimizar} \quad z_i = (c_i - \pi_1 . D) . x_i \tag{I.3.17}$$

$$\text{sujeito a} \quad A_i . x_i = b_i \tag{I.3.18}$$

$$x_i \geq 0 \tag{I.3.19}$$

2 Uma vez obtido x^s, a coluna que entra na base é

$$p_s = \begin{bmatrix} D . x^s \\ \hline 1 \end{bmatrix} \tag{I.3.20}$$

com coeficiente da função objetivo igual a $c.x^s$.

Depois desta breve exposição da forma pela qual é gerada uma coluna que entra na base, podemos resumir o algoritmo de Dantzig e Wolfe:

1 Dada uma solução básica factível para o problema I.3.10 - I.3.12, com base B e multiplicadores $[\pi, \alpha]$, resolver os subproblemas I.3.17 - I.3.19.

2 Calcular $\tilde{c}_s$, usando I.3.16. Se $\tilde{c}_s \geq 0$ então a solução obtida é ótima e a solução do problema original é

$$x^* = \sum_j \lambda_j . x^j, \text{ para } j \text{ básico} \qquad \text{(I.3.21)}$$

onde x^j são os vértices das regiões factíveis dos blocos que correspondem às componentes básicas de λ.

3 Se $\tilde{c}_s < 0$, calcular

$$y = B^{-1} . \begin{bmatrix} A_i . x_i \\ 1 \end{bmatrix} \qquad \text{(I.3.22)}$$

e, através do processo usual de pivoteamento do simplex, obter uma nova inversa da base e um novo conjunto de multiplicadores, voltando ao passo *1*.

A proposta de Rosen

Outro algoritmo que também se baseia em decomposição é atribuído a Rosen [15]. Embora intuitivamente fácil de se entender, este método, como o anterior, está baseado numa série de teoremas que evitaremos citar aqui, dado o caráter expositivo da seção.

Sendo muito breve, podemos dizer que Rosen sugere que se resolva um problema gerando uma sequência de soluções básicas *infactíveis*, mas que correspondem a soluções básicas factíveis para o seu dual.

Para um problema descrito por I.3.1 - I.3.5, considerando-se ainda que as variáveis são canalizadas, isto é,

$$l_1 \leq x_1 \leq u_1, \ldots, l_n \leq x_n \leq u_n \qquad \text{(I.3.23)}$$

o algoritmo consiste em:

o Minimizar cada bloco em separado (sem levar em conta as restrições de estoque).

r Considerando apenas o problema de acoplamento, ou seja, as equações I.3.1, I.3.5 e I.3.23, substituir cada variável básica x_k^B, k = 1, ..., n, por

$$x_k^B = B_k^{-1}.(b_k - N_k.x_k^N) \qquad \text{(I.3.24)}$$

onde B_k é a base do bloco *k*, e N_k a matriz das colunas não básicas do mesmo. Tem-se, então, um novo problema, escrito apenas em função das variáveis não básicas dos blocos:

$$\text{minimizar} \quad \left(\underset{\sim}{c}_1^N\right)^T x_1^N + \ldots + \left(\underset{\sim}{c}_n^N\right)^T x_n \qquad \text{(I.3.25)}$$

$$\text{sujeito a} \quad \underset{\sim}{D}_1^N \; x_1^N + \ldots + \underset{\sim}{D}_n^N \; x_n^N \leq \underset{\sim}{b} \qquad \text{(I.3.26)}$$

$$l_k^N \leq x_k^N \leq u_k^N \;, \; k = 1,\ldots,n \qquad \text{(I.3.27)}$$

onde

$$\left(\underset{\sim}{c}_k^N\right)^T = \left(c_k^N\right)^T - \left(c_k^B\right)^T . B_k^{-1} . N_k \qquad \text{(I.3.28)}$$

$$\underset{\sim}{D}_k^N = D_k^N - D_k^B . B_k^{-1} . N_k \qquad \text{(I.3.29)}$$

$$\underset{\sim}{b} = f - \sum_k D_k^B . x_k^B \qquad \text{(I.3.30)}$$

2 Resolver o problema I.3.25 - 1.3.27 (veja que ele possui, apenas, o mesmo número de linhas que o acoplamento).

3 Calcular, em função de x_k^N, os novos valores de x_k^B, utilizando I.3.24.

Por simplicidade, chamaremos este novo vetor x por x_k^1, em oposição aos valores obtidos na iteração anterior (ou no passo *o*), denominados x_k^o.

4 Verificar se

$$l_k \leq x_k^1 \leq u_k \qquad \text{(I.3.31)}$$

Em caso afirmativo, a solução encontrada é ótima e não é necessário prosseguir.

5 Caso contrário, escolher, para cada bloco q que contenha uma canalização violada, uma variável básica $x_{q_s}^1$ dentre as que não obedecem seus limites, tal que:

$$\theta_{q_s} = \min_j \theta_{q_j} \qquad \text{(I.3.32)}$$

onde, para variáveis que violaram o limite inferior:

$$\theta_{q_j} = \frac{x_{q_j}^0 - l_{q_j}}{x_{q_j}^0 - x_{q_j}^1} \qquad \text{(I.3.33)}$$

e para variáveis acima do limite superior:

$$\theta_{q_j} = \frac{u_{q_j} - x_{q_j}^0}{x_{q_j}^1 - x_{q_j}^0} \qquad \text{(I.3.34)}$$

6 Resolver novamente, para cada bloco q, um programa com as restrições:

$$A_q \cdot x_q = b_q \qquad \text{(I.3.35)}$$

$$l_q \leq x_q \leq u_q \qquad \text{(I.3.36)}$$

e com uma função objetivo que depende do tipo de violação. Assim, se $x^1_{q_j} > u_{q_j}$, obtém-se uma nova base B_q, para o bloco, maximizando x_{q_j}. Se, por outro lado, $x^1_{q_j} < l_{q_j}$, B_q é obtida minimizando x_{q_j}.

Em seguida, voltar ao passo *r*.

Características dos métodos de decomposição

Algumas das características mais relevantes dos métodos de decomposição são:

r Como só se trabalha com problemas pequenos, o consumo de memória costuma ser muito baixo se comparado aos daqueles métodos que trabalham com a matriz tecnológica completa.

2 Além disto, pelo mesmo motivo, a matriz básica utilizada é muito pequena, agilizando o processo de sua

decomposição ou atualização, passo que, normalmente, é o mais demorado em cada iteração do simplex.

3 Finalmente, o desempenho de ambos os métodos será tanto melhor quanto maior for o número de blocos e menor o de restrições de estoque. E nenhum deles costuma funcionar bem quando o número de linhas de acoplamento for razoavelmente grande.

Usando a estratégia das restrições ativas

Outro algoritmo proposto para problema bloco angulares envolve o uso da *estratégia das restrições ativas* dentro do método simplex [1].

Os pontos de que merecem destaque nesta alternativa são a boa estabilidade que ela induz ao processo de resolução dos problemas, o tamanho reduzido da matriz básica e a ausência de variáveis de folga na fase 2 do simplex.

Descrição do algoritmo

Seja x uma solução básica factível não degenerada do problema. Então, o método simplex combinado com a estratégia das

restrições ativas pode ser resumido por:

1 *Teste de Otimalidade.* Verificar se a solução disponível é otima aplicando as condições de Kuhn-Tucker:

$$c + \sum_{i \in I} \pi_i a_i + \sum_{j \in J} \mu_j e_j = 0 \qquad \text{(I.3.37)}$$

onde

I é o conjunto das restrições ativas, e

J o conjunto de variáveis ativas da solução;

a_i é a i-ésima linha ativa da matriz A

e_j é a j-ésima coluna da matriz identidade.

1.1 Para isso, deve-se calcular primeiro os multiplicadores correspondentes a solução, resolvendo

$$B^T.\pi = c_I \qquad \text{(I.3.38)}$$

para obter π (B representa a base), e substituir estes valores em I.3.1 para obter μ:

$$\mu_j = c_j - \sum_{i \in I} \pi_i.a_{i,j}, \quad j \in J \qquad \text{(I.3.39)}$$

1.2 A solução atual, x, será ótima se os multiplicadores atenderem a:

$$\pi_i > 0, \text{ se } a_i.x = b_i^L; \qquad (I.3.40)$$

$$\pi_i < 0, \text{ se } a_i.x = b_i^U; \qquad (I.3.41)$$

$$\mu_j > 0, \text{ se } x_j = l_j; \text{ e} \qquad (I.3.42)$$

$$\mu_j < 0, \text{ se } x_j = u_j \qquad (I.3.43)$$

caso contrário será preciso escolher uma restrição ou variável de multiplicador incorreto para deixar a base.

2 *Calculo de uma direção de busca.* Selecionado um multiplicador incorreto, é preciso determinar a aresta, d, sobre a qual será procurada uma solução de custo menor. Isto é feito obedecendo-se a um dos seguintes passos:

2.1 *Quando o multiplicador escolhido é* π_i. Neste caso, deve-se resolver o sistema

$$B.d = z \qquad (I.3.44)$$

onde

$z = e_i$, se $\pi_i < 0$, e

$z = -e_i$, se $\pi_i > 0$.

2.2 *Quando o multiplicador escolhido é* μ_j. Aqui também será preciso resolver um sistema na forma

$$B.d = z \qquad (I.3.45)$$

onde, agora,

$z = \hat{a}_j$, se $\mu_j > 0$, e

$z = -\hat{a}_j$, se $\pi_j < 0$, e

$\hat{a}_j$ representa a parcela da coluna da variável j correspondente às linhas das restrições ativas.

3 *Determinação do tamanho do passo.* Resta, ainda, para obtermos o novo vértice x^1, calcular a distância deste em relação a x se caminharmos sobre a aresta *d*. Para isto, dada a equação

$$x^1 = x + \lambda.d, \qquad (I.3.46)$$

deve-se adotar como tamanho do passo, λ, o menor valor fornecido pelas equações abaixo:

3.1 Para cada restrição inativa *i*, em relação ao limites inferior e superior, respectivamente:

$$\frac{b_i^U - \sum_j a_{ij}.x}{\sum_j a_{ij}.d_j} \quad e \quad \frac{b_i^L - \sum_j a_{ij}.x}{-\sum_j a_{ij}.d_j} \qquad (I.3.47)$$

3.2 Para cada canalização inativa *j*, em relação aos seus limites superior e inferior, respectivamente:

$$\frac{U_j - x}{d_j} \quad e \quad \frac{L_j - x}{-d_j} \qquad (I.3.48)$$

Peculiaridades interessantes

O relativamente baixo consumo de memória deste método, que pode ser considerado uma de suas virtudes, está associado a dois fatos:

1 A matriz básica utilizada pelo algoritmo é obtida a partir da matriz tecnológica A de I.1.1, extraindo-se desta as linhas correspondentes a restrições não ativas (ou seja, aquelas em que $a_i x \neq b_i$) e as colunas das variáveis ativas (aquelas em que $x_j = l_j$ ou $x_j = u_j$). Observa-se que esta matriz será,

geralmente, muito menor que a base utilizada no método simplex original.

2 Nos problemas em que os diversos blocos compartilham de um mesmo conjunto de recursos e possuem restrições de mesma natureza, como o problema da ração, citado na seção 2 deste capítulo, a matriz tecnológica não precisa ser armazenada. Ao invés disto, guarda-se uma outra matriz que relaciona os insumos às exigências das restrições.

Assim, se todas as rações incluem ingredientes como milho e soja, e todas têm restrições relativas aos níveis de proteína e gordura, por exemplo, os blocos da matriz A terão muitos elementos em comum, que podem ser armazenados numa única matriz e recuperados sempre que se julgar conveniente. Com isso, evita-se o desperdício de guardar os mesmos dados repetidos bloco a bloco.

Outra vantagem que também pode ser atribuída ao tamanho reduzido da base diz respeito ao pequeno tempo consumido na fatoração desta, tornando desnecessário um algoritmo específico para sua atualização, uma vez que é econômico refatorá-la a cada iteração do simplex, eliminando um dos principais focos de instabilidade deste tipo de método.

CAPÍTULO SEGUNDO

DESCRIÇÃO DO ALGORITMO

SEÇÃO 1 OBJETIVO E CARACTERÍSTICAS DO ALGORITMO PROPOSTO

Como foi possível observar no capítulo primeiro, a matriz tecnológica do problema considerado possui algumas características bastante evidentes relativas à sua estrutura:

I A matriz é bastante esparsa, isto é, tem um grande número de elementos nulos;

2 A esparsidade está muito bem organizada, ou seja, a matriz é densa em blocos próximos à diagonal e em uma estreita banda horizontal na parte inferior.

Dada as peculiaridades desta matriz, e uma vez que a base, no método simplex, é tão somente uma submatriz dela, é possível crer que um método para a solução de tais problemas de programação linear que se pretenda eficiente deve levar em conta tal estrutura de forma a obter

1 Uma economia de memória, armazenando apenas os elementos não nulos de todas as matrizes envolvidas e evitando ao máximo o surgimento de novos elementos durante sua execução;

2 Uma economia de tempo, executando apenas as operações que envolvem os elementos não nulos.

Isto mantendo sempre o cuidado de exigir uma certa estabilidade numérica, para que os resultados não fiquem comprometidos.

Com a finalidade de atingir estes objetivos, foi criado um algoritmo que possui as seguintes características gerais:

1 É um algoritmo do tipo simplex com fatoração da base em submatrizes triangulares L e U, tal como idealizado por Bartels e Golub (1969), que provaram suas boas características no tocante à estabilidade;

2 As matrizes oriundas da decomposição da base não ocupam mais espaço do que esta, permitindo um aproveitamento integral de sua boa estrutura;

3 Para evitar uma nova fatoração a cada passo do algoritmo, as substituições de colunas básicas são normalmente feitas por um processo de atualização da base que gera poucos elementos não nulos novos.

Depois desta pequena introdução, passemos mais que depressa à descrição do algoritmo. Não sem antes prevenir, entretanto, ainda a tempo, que, na dúvida entre a prolixidade e a superficialidade, incorreremos mais vezes no primeiro vício.

Vamos considerar um problema de programação linear com variáveis canalizadas na forma

$$\begin{array}{ll} \text{minimizar} & c^T x \\ \text{sujeito a} & Ax = b \\ & 0 \le x \le u \end{array} \qquad \text{(II.2.1)}$$

onde

$c \in \mathbb{R}^n$; $b \in \mathbb{R}^m$; $m < n$; e

a matriz $A \in \mathbb{R}^{m \times n}$ tem posto linha completo.

Passos do algoritmo Simplex

Supondo que já encontramos uma solução básica viável inicial, X, à qual corresponde a base

$$B = [A^B_1 , \ldots , A^B_m] \qquad \text{(II.2.2)}$$

e definindo ainda os vetores

$$X_B = [X_1^B , \ldots , X_m^B] \qquad \text{(II.2.3)}$$

das componentes básicas de X, e

$$X_N = [X_1^N , \ldots , X_{n-m}^N] \qquad \text{(II.2.4)}$$

das demais componentes ou componentes não básicas de X. Então, para obter uma nova solução, X', melhor que X, o método simplex nor recomenda que, a cada passo

1 Encontremos os valores de X_B, resolvendo

$$B.X_B = b \qquad \text{(II.2.5)}$$

uma vez que

$$X_i^N = 0 \quad \text{ou} \quad X_i^N = u_i \qquad \text{(II.2.6)}$$

para todo i = 1, ... , n-m.

2 Formemos o vetor

$$c_B = [c_1^B, \ldots ,c_m^B] \qquad \text{(II.2.7)}$$

das componentes de $\underline{c}$ correspondentes às colunas básicas, e resolvamos

$$B^T.\lambda = c_B \qquad \text{(II.2.8)}$$

(os elementos do vetor λ são conhecidos como multiplicadores do simplex)

3 *Teste de otimalidade.* Para cada coluna não básica A^N_j de A, calculemos

$$\bar{c}^N_j = c^N_j - \lambda^T . A^N_j \qquad \text{(II.2.9)}$$

e verifiquemos se são obedecidas as inequações

$$\bar{c}^N_j > 0, \text{ para } X^N_j = 0, \text{ e} \qquad \text{(II.2.10)}$$

$$\bar{c}^N_j < 0, \text{ para } X^N_j = u^N_j. \qquad \text{(II.2.11)}$$

Em caso afirmativo, a solução atual, *X*, é ótima e não há porque continuar com o processo. Caso contrário, é preciso que escolhamos um índice $\underline{s}$ correspondente a uma variável que não atende a uma das inequações II.2.10 e II.2.11.

4 Resolvamos

$$B.y = A_s \qquad \text{(II.2.12)}$$

5 Encontremos o índice r tal que

$$r = \min \langle r_1, r_2, u_s^N \rangle \qquad \text{(II.2.13)}$$

onde

$$r_1 = \min \left\{ \frac{X_i^B}{|y_i|} \right\} \qquad \text{(II.2.14)}$$

para todo y_i tal que $y_i.\delta > 0$;

$$r_2 = \min \left\{ \frac{u_i^B - X_i^B}{|y_i|} \right\} \qquad \text{(II.2.15)}$$

para todo y_i tal que $y_i.\delta < 0$; e

$$\delta = \left\{ \begin{array}{ll} 1, & \text{se } X_s^N = 0 \\ -1, & \text{se } X_s^N = u_s^N \end{array} \right\} \qquad \text{(II.2.16)}$$

6 Retiremos a coluna A_r^B de B e acrescentemos A_s formando uma nova base, podendo, então, retornar ao passo *3*.

Passos dos quais depende a eficiência do Simplex

O bom desempenho do algoritmo simplex tal como ele está sendo apresentado irá depender, basicamente, da eficiência com que são resolvidos os sistemas

$$B.X_B = b, \qquad B^T.\lambda = c_B, \qquad B.y = A_S$$

que aparecem nos supracitados passos 1, 2 e 4, respectivamente, e também a substituição de colunas de B, o que chamamos de *atualização da base* (e que aparece no passo 6).

A partir da próxima seção, vamos nos preocupar tão somente em detalhar e analisar as conseqüências sobre o desempenho advindas da manutenção da base, no simplex, decomposta em submatrizes triangulares, o que representamos por

$$B.P = L.U \qquad (II.2.17)$$

onde L e U são, respectivamente, matrizes triangulares inferior e superior, e P é uma matriz de permutações de linhas.

A decomposição de uma matriz quadrada qualquer, B, em submatrizes triangulares é freqüentemente empregada para tornar mais rápido o processo de resolução de uma seqüência de sistemas lineares do tipo

$$B.\ x = \beta, \quad \text{ou} \quad B^T.x = \gamma \qquad \text{(II.3.1)}$$

que diferem entre si apenas pelos elementos de β e γ. Isto porque o uso de matrizes triangulares transforma este processo numa sucessão trivial de substituições.

Para matrizes que não possuem uma estrutura determinada e para matrizes densas, o processo costumeiro da decomposição LU consiste em fazer

$$P.B = L.U \qquad \text{(II.3.2)}$$

onde

B tem dimensão m x m;

L (triangular inferior),é o produto de matrizes triangulares unitárias;

U é triangular superior; e

P é o produto de matrizes de permutações de linhas.

De forma mais explícita:

$$L = L_1 L_2 L_3 \ \ldots \ L_{mxm-m'} \qquad \text{(II.3.3)}$$

$$P = P_m \ \ldots \ P_3 P_2 P_1 \qquad \text{(II.3.4)}$$

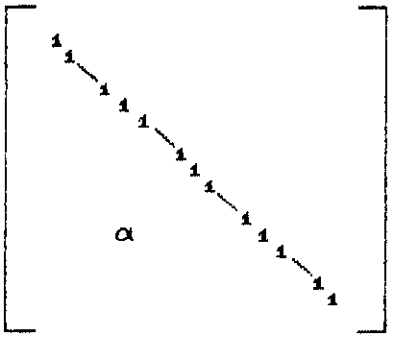


Fig II.3.1 - Uma matriz triangular unitária

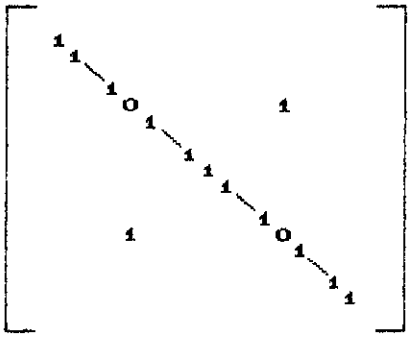


Fig II.3.2 - Uma matriz de permutação de linhas

Nas figuras II.3.1 e II.3.2 são exemplificadas as matrizes que compõem L e P. Assim, se a matriz L_i foi utilizada para zerar o elemento b_{jk} de B, então terá apenas um elemento não nulo — $l_{jk} = \alpha$ — abaixo da diagonal principal. Por sua vez, se P_i é a matriz utilizada para permutar as linhas *j* e *k* de B, então pode ser montada a partir da identidade, trocando-lhe as posições das colunas *j* e *k*.

Neste tipo de decomposição, a matriz de permutações de linhas aparece não somente para garantir que os elementos da diagonal sejam não nulos (requisito necessário para que o problema possa ser resolvido), mas também para conferir ao método uma maior estabilidade numérica, evitando erros de arredondamento que possam comprometer a obtenção da solução.

A estratégia de permutação utilizada geralmente para manter a estabilidade é chamada de *pivotamento parcial* e consiste em permutar linhas (ou colunas) de forma que, a cada passo, o elemento que irá ficar na diagonal principal da matriz (ou seja, o pivô) seja aquele de maior valor absoluto dentre os candidatos.

Mas, para resolver sistemas em que a matriz tem forma bloco-angular, a estratégia de pivoteamento parcial com permutações de linhas tal como descrita acima é extremamente desaconselhável, uma vez que pode gerar um enorme número de elementos não nulos novos durante a fatoração.

Basta imaginar um caso em que seja preciso permutar uma linha do acoplamento com outra de um bloco, como na situação extrema mostrada abaixo:

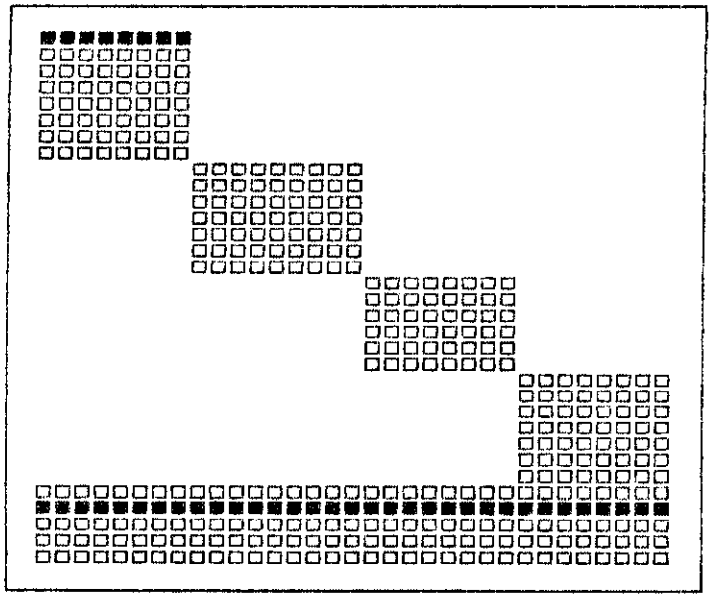


Fig II.3.3 - A base antes
do início da fatoração

Se, dada a base da figura II.3.3, precisássemos permutar, no primeiro passo da fatoração, as linhas destacadas, o resultado seria o aparecimento de um grande número de elementos novos em posições indesejadas, mostrados em destaque na figura II.3.4.

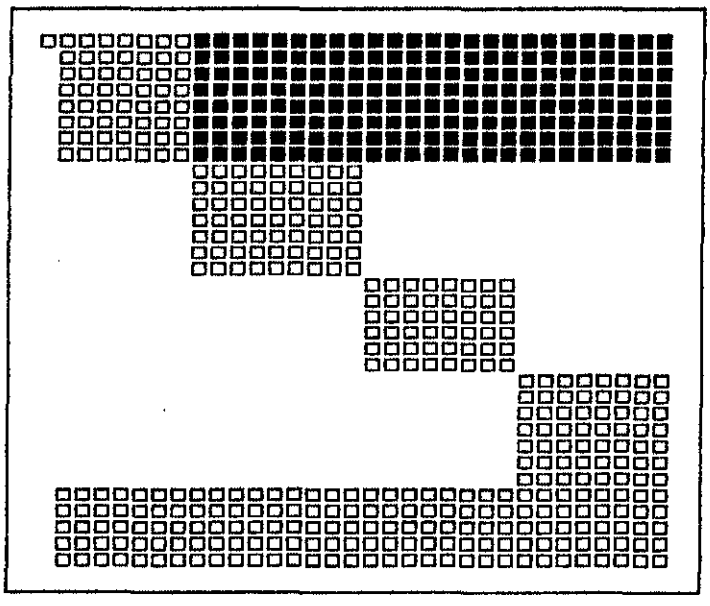


Fig II.3.4 - A matriz resultante da eliminação dos elementos abaixo da diagonal da coluna 1 da base.

Características da fatoração proposta

Com a finalidade de manter intacta a estrutura da base (isto é, o número e a posição dos seus elementos não nulos) sem, entretanto, abrir mão da estabilidade numérica, optamos por manter o pivoteamento parcial, fazendo, contudo, permutações de colunas e não de linhas. Com isso, passamos a ter, pelo contrário

$$B.P = L.U \qquad \text{(II.3.5)}$$

com

L triangular inferior;

$U = U_{m\times m-m} \ \ldots \ U_3U_2U_1$ (produto de matrizes triangulares superiores unitárias); e

$P = P_1P_2P_3 \ \ldots \ P_m$.

Essa mudança não provoca nenhuma alteração na sistemática da fatoração. Apenas fornece matrizes ligeiramente diferentes.

Além disso, a fatoração invertida tal como pretendemos fazer possui as seguintes vantagens:

1 A quantidade de memória gasta para armazenar os elementos de L e U somadas é mínima, ou seja, igual àquela gasta para armazenar a base;

2 Mais que isso, os elementos de L e U podem ser guardados exatamente nas mesmas posições dos elementos de B, facilitando a manipulação dos dados;

E é claro que com esta economia de memória, o tempo de fatoração e de solução de sistemas também torna-se mínimo, e a estabilidade numérica se preserva, não só pelo uso do

pivoteamento parcial, como pelo fato de que o erro também costuma ser proporcional ao número de elementosda matriz.

Assim, depois de fatorada e já considerando as permutações de colunas, a base terá o segunte aspecto

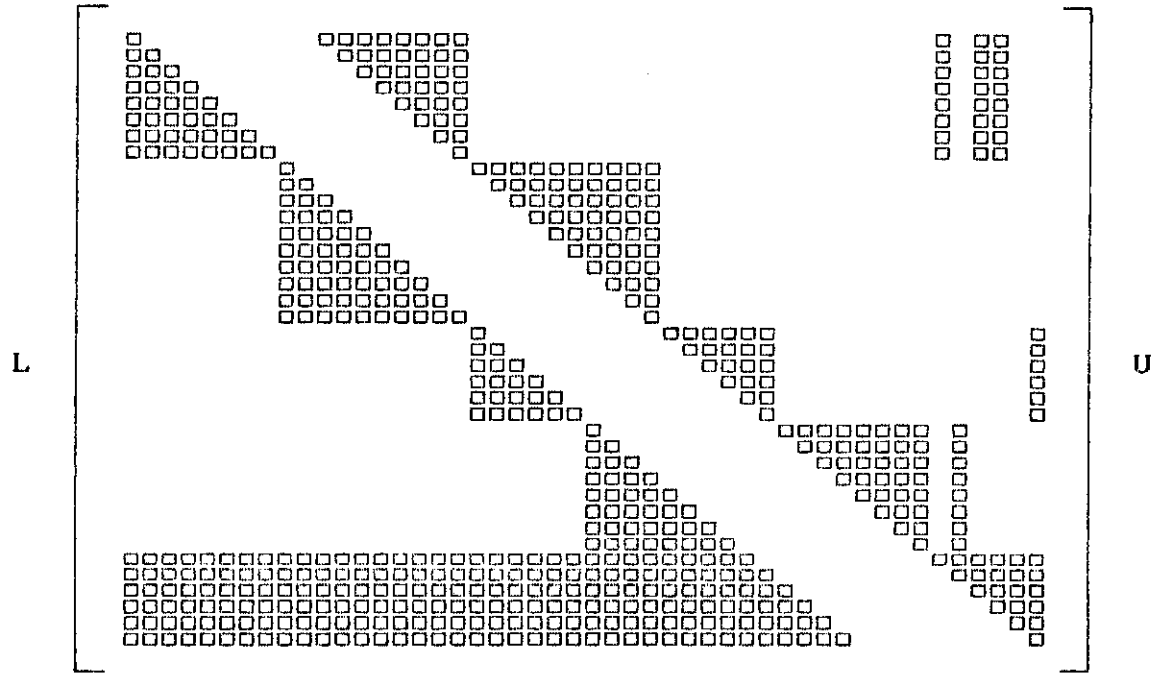


Fig II.3.5 - As submatrizes triangulares oriundas da fatoração

Como se pode ver na figura II.3.5, o número de elementos não nulos de L e U somados é igual ao de B. Os elementos que aparecem nas colunas mais a direita da matriz U são provenientes dos blocos não quadrados da base e não foram criados na fatoração, mas tão somente movidos para estas

posições quando das permutações de colunas necessárias para obter os pivôs correspondentes às linhas do acoplamento.

Pode-se reparar, inclusive, que se estes elementos forem agrupados com os blocos correspondentes, ter-se-á a mesma disposição dos elementos da matriz B original.

Detalhando a fatoração LU com o auxílio de um exemplo numérico

O método usado na decomposição da base é o conhecido processo de *eliminação de Gauss*, que vamos apresentar brevemente. No exemplo mostrado abaixo, utilizaremos uma representação compacta das matrizes oriundas da fatoração, em que as matrizes L e U são apresentadas ao mesmo tempo como uma só matriz. Neste tipo de representação, os elementos acima da diagonal pertencem a U, enquanto os demais correspondem à matriz L. Todos os elementos da diagonal de U são iguais a 1, dispensando a sua representação.

Vamos supor que queiramos fatorar a matriz da figura II.3.6:

1.00	2.00	5.00	-3.00						
2.00	-1.00	7.00	2.00						
				1.00	3.00				
				-1.00	1.00				
						1.00	-1.00	-3.00	2.00
						1.00	1.00	-1.00	3.00
-1.00	2.00	-1.00	3.00	2.00	1.00	-2.00	1.00	3.00	-1.00
2.00	-3.00	-2.00	-1.00	1.00	-3.00	-4.00	5.00	-1.00	1.00
3.00	1.00	1.00	1.00	1.00	2.00	1.00	-1.00	3.00	1.00
1.00	-1.00	-1.00	2.00	2.00	-1.00	3.00	-1.00	-1.00	1.00

Fig. II.3.6 - Um exemplo de base, B, com estrutura bloco-angular

No nosso problema, estamos preocupados em transformar a base numa matriz triangular inferior L. Para isso é preciso eliminar os elementos acima da diagonal de B, o que é feito através dos seguintes passos:

1.1 Inicialmente, escolhemos o primeiro pivô como sendo aquele de maior valor absoluto dentre todos os elementos da primeira linha (no caso, o elemento b_{13}) e permutamos as colunas 1 e 3 para fazer com que o pivô passe para a diagonal.

Com isso, a primeira coluna de L está formada e não mais sofrerá qualquer alteração. A matriz torna-se, então:

5.00	2.00	1.00	-3.00						
7.00	-1.00	2.00	2.00						
				1.00	3.00				
				-1.00	1.00				
						1.00	-1.00	-3.00	2.00
						1.00	1.00	-1.00	3.00
-1.00	2.00	-1.00	3.00	2.00	1.00	-2.00	1.00	3.00	-1.00
-2.00	-3.00	2.00	-1.00	1.00	-3.00	-4.00	5.00	-1.00	1.00
1.00	1.00	3.00	1.00	1.00	2.00	1.00	-1.00	3.00	1.00
-1.00	-1.00	1.00	2.00	2.00	-1.00	3.00	-1.00	-1.00	1.00

Fig. II.3.7 - A mesma matriz, após permutadas as colunas 1 e 3. A primeira coluna (destacada) já pertence à matriz L.

Como se pode ver, qualquer que fosse o pivô escolhido, o número de elementos não nulos jamais aumentaria. A matriz P_1, referente à permutação executada na determinação do primeiro pivô, é semelhante à identidade, diferindo desta apenas pelo fato das colunas 1 e 3 terem sido trocadas.

1.2 Depois de permutadas as colunas, é preciso eliminar os elementos acima da diagonal na linha 1 da matriz. Para o

elemento b_{12}, por exemplo, isto é feito multiplicando-se a primeira coluna por

$$-b_{12}/b_{11} \qquad \text{(II.3.6)}$$

e somando-se esta à coluna 2. O termo da fórmula II.3.6 corresponde ao elemento u_{12} da matriz U.

Este processo é repetido para as demais colunas com elementos não nulos na primeira linha (sempre usando a coluna do pivô), de forma que, ao final do primeiro passo, temos:

5.00	-0.40	-0.20	0.60						
7.00	-3.80	0.60	6.20						
				1.00	3.00				
				-1.00	1.00				
						1.00	-1.00	-9.00	2.00
						1.00	1.00	-1.00	3.00
-1.00	2.40	-0.80	2.40	2.00	1.00	-2.00	1.00	3.00	-1.00
-2.00	-2.20	1.60	-2.20	1.00	-3.00	-4.00	5.00	-1.00	1.00
1.00	0.60	2.80	1.60	1.00	2.00	1.00	-1.00	3.00	1.00
-1.00	-0.60	1.20	1.40	2.00	-1.00	9.00	-1.00	-1.00	1.00

Fig. II.3.8 - A matriz ao final do primeiro passo da fatoração. Em destaque, a primeira coluna de L e a primeira linha de U.

Na matriz acima, já se pode ver a primeira linha de U. Os elementos desta linha poderão ser permutados, mas manterão seus valores até o final da fatoração.

2.1 Passamos, então, para a segunda iteração do algoritmo, escolhendo, dentre os elementos restantes da matriz B (aqueles que ainda não pertencem a L ou a U), o de maior valor absoluto para ser o pivô. É fácil deduzir da figura II.3.8 que o elemento b_{24}, cujo valor é 6,20, deve ser o escolhido. A seguir as colunas 2 e 4 são permutadas, e a segunda coluna de L está criada (figura II.3.9).

5.00	0.60	-0.20	-0.40					
7.00	6.20	0.60	-3.80					
				1.00	3.00			
				-1.00	1.00			
						1.00	-1.00	-3.00	2.00
						1.00	1.00	-1.00	3.00
-1.00	2.40	-0.80	2.40	2.00	1.00	-2.00	1.00	3.00	-1.00
-2.00	-2.20	1.60	-2.20	1.00	-3.00	-4.00	5.00	-1.00	1.00
1.00	1.60	2.80	0.60	1.00	2.00	1.00	-1.00	3.00	1.00
-1.00	1.40	1.20	-0.60	2.00	-1.00	3.00	-1.00	-1.00	1.00

Fig. II.3.9 - A matriz após terem sido permutadas as colunas 2 e 4. Já se pode ver formada a segunda coluna de L.

2.2 Resta, nesta iteração, eliminar os elementos acima da diagonal na linha 2 da matriz. Novamente isto é feito com o auxílio da coluna do pivô (que é somada a cada coluna j depois de multiplicada por $-b_{2j}/b_{22}$). Com isso, ao final do passo 2 da fatoração temos:

5.00	0.60	-0.20	-0.40						
7.00	6.20	-0.10	0.61						
				1.00	9.00				
				-1.00	1.00				
						1.00	-1.00	-9.00	2.00
						1.00	1.00	-1.00	9.00
-1.00	2.40	-1.09	3.87	2.00	1.00	-2.00	1.00	3.00	-1.00
-2.00	-2.20	1.81	-9.55	1.00	-9.00	-4.00	5.00	-1.00	1.00
1.00	1.60	2.64	1.58	1.00	2.00	1.00	-1.00	3.00	1.00
-1.00	1.40	1.06	0.26	2.00	-1.00	9.00	-1.00	-1.00	1.00

Fig. II.9.10 - Terminada a fatoração do primeiro bloco.

Podemos dar por encerrada, então, a fatoração do primeiro bloco da matriz. Pode-se observar que, uma vez que o bloco não é quadrado, algumas de suas colunas terão que ser permutadas antes de passarmos para o próximo passo. Estas colunas voltarão a participar da decomposição quando estivermos tratando das linhas do acoplamento.

O processo de fatoração do segundo e do terceiro blocos é semelhante ao do primeiro e, por isso, nos absteremos de repetir os mesmos enfadonhos passos descritos acima. Basta notar que não haverá nenhuma interferência de um bloco na fatoração de outro.

Por fim, teremos a seguinte matriz:

5.00	0.60					-0.20	-0.40		
7.00	6.20					-0.10	0.61		
		9.00	-0.99						
		1.00	-1.33						
				-9.00	0.67			0.99	-0.99
				-1.00	2.99			-0.29	-0.57
-1.00	2.40	1.00	1.67	9.00	1.00	-1.09	9.87	-1.29	-0.57
-2.00	-2.20	-9.00	2.00	-1.00	0.99	1.81	-9.55	-4.49	5.14
1.00	1.60	2.00	0.99	9.00	9.00	2.64	1.58	1.13	-9.71
-1.00	1.40	-1.00	2.99	-1.00	0.99	1.06	0.26	2.57	-0.86

Fig. II.9.11 - A matriz após terminada a fatoração dos blocos.

Como se observa na figura II.3.11, as colunas que sobraram nos blocos não quadrados da base foram transferidas para o lado direito da matriz, sem, contudo, fazer com que o número de elementos não nulos aumentasse.

Para terminar a fatoração, resta, então, decompor a parte da matriz que envolve apenas as restrições de estoque, bastando para isso repetir o mesmo processo utilizado nos passos anteriores.

As matrizes L e U resultantes ao final da fatoração são fornecidas na figura II.3.12.

5.00	0.60					-0.40		-0.20	
7.00	6.20					0.61		-0.10	
		3.00	-0.33						
		1.00	-1.99						
				-3.00	0.67		0.33		-0.33
				-1.00	2.33		-0.29		-0.57
-1.00	2.40	1.00	1.67	3.00	1.00	3.87	0.93	0.27	0.15
-2.00	-2.20	-3.00	2.00	-1.00	0.33	-3.55	-5.61	0.90	0.82
1.00	1.60	2.00	0.33	3.00	3.00	1.58	1.67	3.56	0.59
-1.00	1.40	-1.00	2.33	-1.00	0.33	0.26	2.66	1.92	2.51

Fig. II.3.12 - As matrizes L e U após terminada a fatoração

As matrizes das permutações citadas podem ser agrupadas formando uma única matriz P, ilustrada na figura II.3.13.

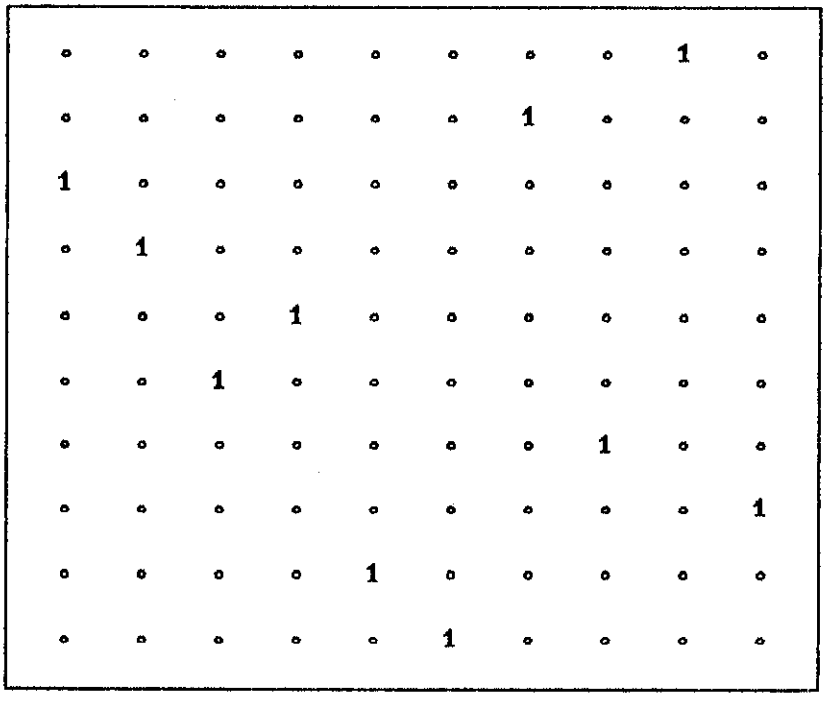


Fig. II.3.13 - A matriz de permutações P.

SEÇÃO 4 A ATUALIZAÇÃO DA BASE

O passo do algoritmo simplex referente à substituição de uma coluna básica por outra coluna da matriz tecnológica do problema — a *atualização da base* — corresponde a obter, a partir de B, uma nova matriz B' tal que:

$$B' = B + (A_s - B_r).e_r^T \qquad \text{(II.4.1)}$$

onde

B' é a nova base;

A_s é a coluna que entra na base;

B_r é a coluna que sai da base; e

e_r é a coluna da matriz identidade que tem 1 na posição r.

A atualização de Bartels e Golub

O processo de atualização envolvendo permutações de linhas descrito por Bartels [2] é bastante simples e consiste em

1 Encontrar um vetor $\underline{v}$ tal que

$$L.v = A_s \qquad \text{(II.4.2)}$$

para que se possa escrever o termo entre parêntesis na equação II.4.1 em função da matriz L, uma vez que a coluna que sai da base é dada por:

$$B_r = L.U_r. \qquad \text{(II.4.3)}$$

Supondo, por simplificação, que não fizemos nenhuma permutação durante a fatoração, então

$$B = L.U \qquad (II.4.4)$$

E, assim, teremos

$$B' = L.U + (L.v - L.U_r).e_r^T, \text{ ou} \qquad (II.4.5a)$$

$$B' = L.\left[U + (v - U_r).e_r^T \right] \qquad (II.4.5b)$$

2 Tornar $U' = U + (v - U_r).e_r^T$ uma matriz Hessenberg superior (triangular superior a menos de uma parte da diagonal imediatamente abaixo da diagonal principal):

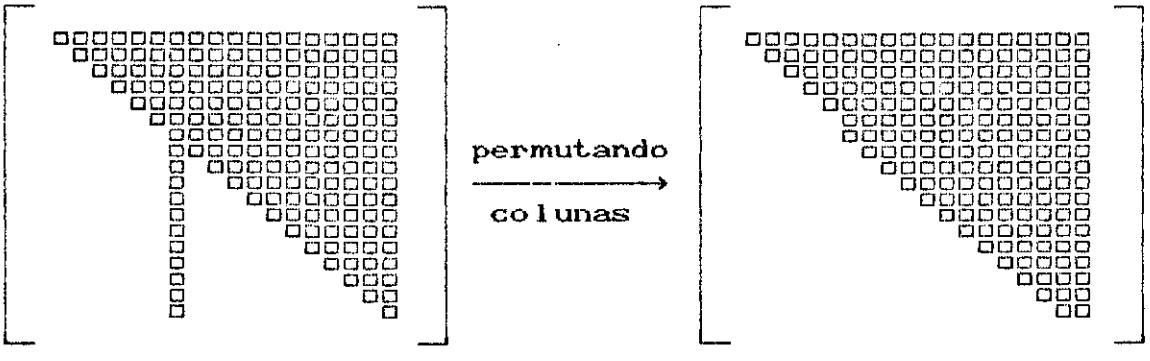


Fig. II.4.1 - a) A matriz U' apresentando elementos não nulos abaixo da diagonal na coluna r. b) a mesma matriz na forma Hessenberg superior.

3 Finalmente, triangularizar U' mediante permutações de linhas e transformações com matrizes triangulares unitárias que se agregam a L pela direita. Donde teremos

$$U' = P_1^{-1}.L'_1.P_2^{-1}.L'_2 \ \ldots \ P_m^{-1}.L'_m.\hat{U} \qquad \text{(II.4.6)}$$

e se

$$P.B' = L.P_1^{-1}.L'_1.P_2^{-1}.L'_2 \ \ldots \ P_m^{-1}.L'_m.\hat{U} \qquad \text{(II.4.7)}$$

então, substituindo

$$\tilde{L} = L.L'_1.L'_2 \ \ldots \ L'_m \quad \text{e} \qquad \text{(II.4.8)}$$

$$\hat{P} = P_m \ \ldots \ P_2.P_1.P \qquad \text{(II.4.9)}$$

obtemos

$$\hat{P}.B' = \tilde{L}.\hat{U} \qquad \text{(II.4.10)}$$

e podemos passar ao próximo ciclo do método simplex.

A atualização proposta

Na nossa decomposição de B, fizemos com que a matriz U fosse um produto de matrizes triangulares unitárias (e não L).

Por esse motivo, devemos procurar uma equação para B' na qual seja possível isolar U (e não L como na atualização de Bartels e Golub). Assim:

r Primeiro, encontramos ω tal que

$$\omega^T.U = e_r^T \qquad \text{(II.4.11)}$$

e calculamos a diferença entre a coluna que entra e aquela que sai da base:

$$\nu = A_s - B_r \qquad \text{(II.4.12)}$$

Com isso, onde tínhamos

$$B' = B + (A_s - B_r).e_r^T \qquad \text{(II.4.13)}$$

passamos a ter

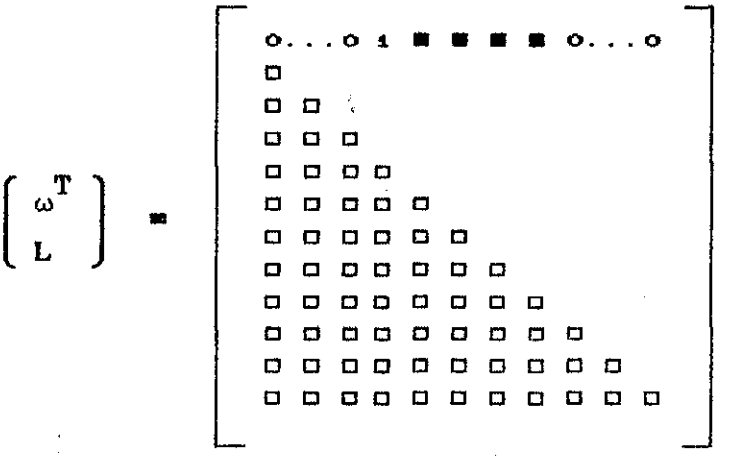
$$B' = L.U + \nu .\omega^T.U, \quad \text{ou} \qquad \text{(II.4.14)}$$

$$B' = (L + \nu.\omega^T) .U \qquad \text{(II.4.15)}$$

Esta última equação pode ser escrita de outra forma como sendo

$$B' = (\nu\ I).\begin{pmatrix} \omega^T \\ L \end{pmatrix} .U \qquad \text{(II.4.16)}$$

onde a matriz $\begin{bmatrix} \omega^T \\ L \end{bmatrix}$ é triangular inferior a menos da primeira linha, que possui elementos não nulos a partir da coluna r, como se vê na figura II.4.2.



- Fig. II.4.2 -

2 Em seguida, zeramos os elementos da primeira linha dessa matriz, utilizando um processo de eliminação de Gauss com permutações de colunas semelhante ao que foi exposto na seção anterior (mais adiante, descreveremos com maior profundidade o critério usado nas permutações), de forma que

$$\begin{bmatrix} \omega^T \\ L \end{bmatrix} .P^* = L^*.U^* \quad \text{onde} \quad L^* = \begin{bmatrix} \rho . e_k^T \\ \tilde{L} \end{bmatrix} \qquad \text{(II.4.17)}$$

e:

. k é a única posição não nula de ω após a eliminação;

$\tilde{L}$ é triangular inferior a menos da coluna k;

P^* é uma matriz de permutações; e

U^* é triangular superior.

A matriz U^* apresentada acima, ao contrário de $\tilde{L}$, não é mantida na forma triangular, mas antes como um produto de matrizes unitárias. Desta forma, se U_1^* é a matriz utilizada na primeira eliminação de um elemento não nulo da matriz da figura II.4.2, e se, da mesma maneira, U_j^* é a matriz triangular unitária empregada na j-ésima eliminação de um total de m-r+1 elementos não nulos existentes no vetor ω^T, onde m é o índice do último elemento não nulo de ω, então

$$U^* = U^*_{m-r}. \ \ldots \ .U^*_j. \ \ldots \ .U^*_2.U^*_1 \qquad \text{(II.4.18)}$$

Analogamente, a matriz P^* também pode ser entendida como um produto de matrizes, cada qual representando apenas uma permutação, de forma que:

$$P^* = P^*_1.P^*_2. \ \ldots \ . P^*_j. \ \ldots \ .P^*_{m-r} \qquad \text{(II.4.19)}$$

A eliminação dos elementos de ω é feita primeiro zerando ω_{m-1} em função de ω_m[1], ou seja, somando à coluna m-1 a coluna m multiplicada pelo termo

---

[1]pode-se optar pela operação inversa, se seguida da permutação das colunas.

$$u^{*}_{m-1,m} = -\omega_{m-1}/\omega_{m} \qquad \text{(II.4.20)}$$

Com isso, passamos a ter uma nova coluna m-1 em L:

$$\tilde{L}_{m-1} = L_{m-1} + u^{*}_{m-1,m}.L_{m} \qquad \text{(II.4.21)}$$

e uma nova matriz triangular unitária, U^{*}_{m-1}, que tem $u^{*}_{m-1,m}$ como único elemento não nulo fora da diagonal (exatamente na linha m-1 e na coluna m):

$$U^{*}_{m-1} = I + e_{m-1}.u^{*}_{m-1,m}.e^{T}_{m} \qquad \text{(II.4.22)}$$

Eliminado o elemento ω_{m-1}, passa-se à coluna m-2 e assim sucessivamente até chegarmos à primeira coluna não nula de ω.

3 Depois de obtida a matriz L^{*}, podemos voltar a escrever a equação de B' na forma original:

$$B' = (\nu\ I).\begin{pmatrix} \rho.e^{T}_{k} \\ \tilde{L} \end{pmatrix}.U^{*}.P^{*}.U = (\nu.\rho.e^{T}_{k} + \tilde{L}).U^{*}.P^{*}.U \qquad \text{(II.4.23)}$$

Adicionamos, então, $\nu.\rho.e^{T}_{k}$ a $\tilde{L}$ formando $L^{\#}$ que é triangular inferior a menos da coluna k.

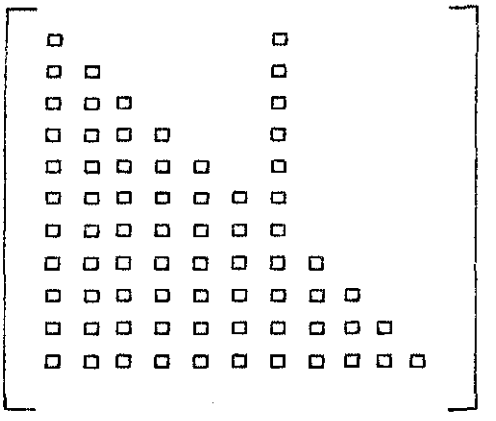


Fig. II.4.3 - A matriz $L^{\#}$

4 Por fim, eliminamos os elementos acima da diagonal da coluna k de $L^{\#}$ com novo processo de eliminação e novas permutações de colunas, gerando outras matrizes triangulares $\hat{U}$ e de permutação $\hat{P}$, de forma que:

$$L^{\#} = \hat{L}.\hat{U}.\hat{P} \qquad \text{(II.4.24)}$$

$$B' = L^{\#}.U^{*}.P^{*}.U = \hat{L}.\hat{U}.\hat{P}.U^{*}.P^{*}.U \qquad \text{(II.4.25)}$$

Aqui, para tornar $L^{\#}$ triangular, eliminamos, inicialmente, o primeiro elemento não nulo da coluna k, $l^{\#}_{p,k}$, fazendo

$$\hat{L}_k = L^{\#}_k + \hat{u}_{p,k} L^{\#}_p \qquad \text{(II.4.26)}$$

onde

$$\hat{u}_{p.k} = -l^{\#}_{p,k} / l^{\#}_{k,k} \qquad \text{(II.4.27)}$$

Com isso, passamos a ter também uma nova matriz $\hat{U}_p$, que é triangular unitária com o elemento $\hat{u}_{p,k}$ na linha p e na coluna k (onde k > p).

Segue-se, então, a eliminação dos elementos das linhas p+1 até k-1 da coluna k de $L^{\#}$.

Terminado este passo, B' será a nova base a ser utilizada pelo simplex. Se for preciso, mais tarde, atualizá-la novamente, isto será feito agregando novas matrizes triangulares unitárias e de permutação à esquerda de $\hat{U}$.

Estratégias de pivoteamento

Nos passos dois e quatro do algoritmo descrito acima, foi preciso eliminar alguns elementos para tornar novamente as matrizes triangulares. Em ambos os casos, repetimos o processo de eliminação da fatoração, embora com algumas sutis diferenças.

A principal delas decorre do fato de que durante a fatoração foi possível empregar uma estratégia de pivoteamento com permutação de colunas, já que isso não implicava num aumento

da memória necessária para armazenar as matrizes L e U. Mas, na atualização, o pivoteamento parcial pode não ser aconselhável em operações que envolvem colunas de blocos diferentes. Donde teremos que fazer concessões à estabilidade para que o número de elementos não nulos de L não cresça demasiadamente.

Analisaremos isto nos dois casos em que eliminamos elementos na atualização:

A eliminação existente no passo 2

No passo 2 temos uma matriz com uma linha a mais que o número de colunas (fig. II.4.2), e precisamos zerar elementos desta linha. Podemos distinguir, aqui, duas situações diferentes:

Caso 1: As colunas que participarão da eliminação, k e m, pertencem ao mesmo bloco da base (veja a figura II.4.4).

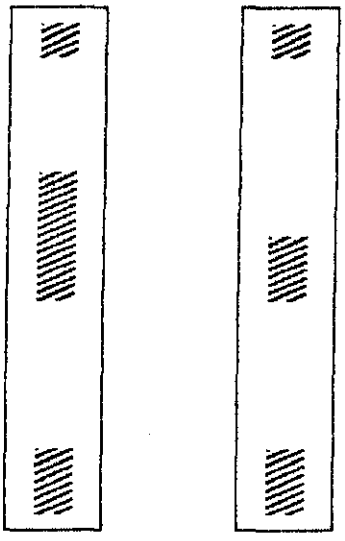


Fig. II.4.4
a) Coluna K;
b) Coluna m;

Aqui, a melhor forma de eliminar o primeiro elemento da coluna $\underline{k}$ consiste em fazer:

$$\tilde{L}_k = L_k - \frac{\omega_k}{\omega_m} \cdot L_m \qquad \text{(II.4.28)}$$

embora o contrário (eliminar o elemento da coluna $\underline{m}$ e em seguida permutar as colunas) não seja muito ruim.

Caso 2: As colunas $\underline{k}$ e $\underline{m}$ pertencem a blocos diferentes da base, como mostra a figura II.4.5.

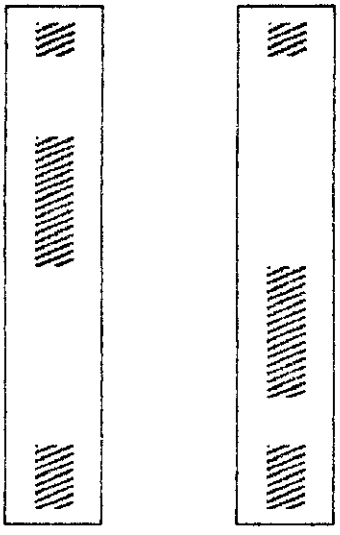


Fig. II.4.5
a) Coluna K;
b) Coluna m;

Neste caso, fazer

$$\tilde{L}_k = L_k - \frac{\omega_k}{\omega_m} \cdot L_m \qquad \text{(II.4.29)}$$

poderia ocasionar uma aumento permanente do número de elementos não nulos de L, enquanto o processo oposto, ou seja, fazer

$$\tilde{L}_m = L_m - \frac{\omega_m}{\omega_k} \cdot L_k \qquad \text{(II.4.30)}$$

provocaria, provavelmente, apenas um pequeno aumento do número de elementos de U, sem reflexos nas atualizações posteriores.

Mas para este tipo de eliminação, é difícil concluir qual deve ser a estratégia a ser usada. É preciso, antes, levar em conta a freqüência com que ocorre cada caso acima descrito, e as conseqüências em eliminações posteriores, do uso contínuo de um relaxamento do pivotamento parcial.

Uma boa sugestão é, na dúvida, optar por preservar a estabilidade e esquecer a esparsidade, considerando que seria pequena a eficácia relativa de um método que tentasse reduzir bastante o número de elementos não nulos criados neste passo.

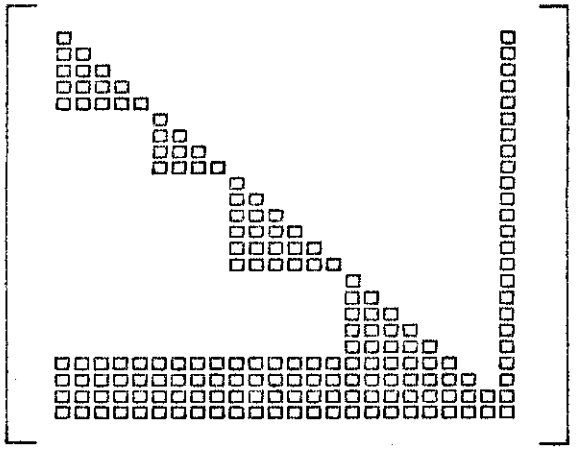
Neste caso, escolhemos a equação II.4.30 se $\omega_k \geq \omega_m$ e a equação II.4.29 se $\omega_k < \omega_m$.

Por outro lado, de forma geral, o uso da equação II.4.30 produz matrizes ligeiramente mais esparsas, o que pode sugerir uma estratégia que a privilegie.

No próximo capítulo são descritos alguns resultados experimentais, os quais foram determinantes na escolha da estratégia neste passo, no programa de computador elaborado.

A eliminação do passo 4

No passo 4, a matriz que queremos tornar triangular, $L^{\#}$, possui uma estrutura tal como mostrada na figura II.4.6.



- Fig II.4.6 -

Imagine que, para eliminar os elementos acima da diagonal da coluna k desta matriz, seja necessário, se quisermos usar pivoteamento parcial, fazer

$$\hat{L}_i = L^{\#}_i - \frac{l^{\#}_{k,i}}{l^{\#}_{k,k}} \cdot L^{\#}_k \qquad \text{(II.4.31)}$$

seguido de uma permutação das colunas i e k, para $i = p,\ldots,k-1$, onde p é a primeira linha com elemento não nulo na coluna k.

A conseqüência seria a obtenção de uma matriz triangular totalmente densa, ou seja, com todos os elementos abaixo da diagonal diferentes de zero, o que é extremamente inconveniente e pode encarecer demasiadamente a atualização,

tanto do ponto de vista do tempo quanto da memória necessários para a sua execução, tornando-a inviável.

Por outro lado, se fizermos

$$\hat{L}_k = L^{\#}_k - \frac{l^{\#}_{k,k}}{l^{\#}_{k,i}} \cdot L^{\#}_i \qquad \text{(II.4.32)}$$

então não haveria nenhum acréscimo a $\hat{L}$.

Por esse motivo, preferiu-se adotar, aqui, uma outra estratégia.

Chamando

$$\mu_i = - \frac{l^{\#}_{k,k}}{l^{\#}_{k,i}} \qquad \text{(II.4.33)}$$

utilizamos II.4.32 sempre que

$$\mu_i \leq \bar{\mu}, \qquad \text{(II.4.34)}$$

onde $\bar{\mu}$ é um limitante estipulado para μ (normalmente se utiliza $\bar{\mu}$= 10).

E para não comprometer em demasia a estabilidade, emprega-se II.4.31 quando μ_i não satisfizer II.4.34.

Um exemplo numérico da atualização

Vamos aproveitar a matriz B do exemplo da fatoração (fig. II.3.4) para a qual já dispomos das matrizes L e U (fig. II.3.10) para mostrar em detalhe como é executada a atualização.

Assim, vamos supor que queiramos substituir a coluna 7 de B, por A_s, dada abaixo

$$A_s^T = \begin{bmatrix} 0 & 0 & 3 & 1 & 0 & 0 & 1 & -2 & -2 & 3 \end{bmatrix}$$

Os passos necessários à obtenção da nova base decomposta são:

1.1 Em primeiro lugar, resolver o sistema

$$U^T.\omega = e_7 \qquad \text{(II.4.35)}$$

o que fornece

$$\omega^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0.2972 & 0.9996 \end{bmatrix}$$

1.2 Em seguida, é preciso calcular $v = A_s - B_7$:

$$\begin{bmatrix} 0 \\ 0 \\ 3 \\ 1 \\ -1 \\ -1 \\ 3 \\ 2 \\ -3 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 3 \\ 1 \\ 0 \\ 0 \\ 1 \\ -2 \\ -2 \\ 3 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ -2 \\ -4 \\ 1 \\ 3 \end{bmatrix}$$

ν $\quad A_S \quad B_7$

1.3 Resta, então, neste passo, montar a matriz $\begin{bmatrix} \omega^T \\ L \end{bmatrix}$.

Esta matriz é mostrada na figura II.4.7:

0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.90	1.00
5.00									
7.00	6.20								
		9.00							
		1.00	-1.99						
				-3.00					
				-1.00	2.99				
-1.00	2.40	1.00	1.67	3.00	1.00	3.87			
-2.00	-2.20	-3.00	2.00	-1.00	0.99	-3.55	-5.61		
1.00	1.60	2.00	0.93	3.00	3.00	1.58	1.67	3.56	
-1.00	1.40	-1.00	2.99	-1.00	0.99	0.26	2.66	1.92	2.51

- Fig. II.4.7 -

2 O segundo passo consiste em eliminar os elementos da primeira linha da matriz da figura acima.

Vamos adotar, aqui, por simplicidade, a estratégia de pivotamento descrita em II.4.30 sempre que os elementos acrescidos a U forem menores que o termo limitante $\eta = 10$, semelhante ao que foi descrito na equação II.4.34. Portanto:

2.1 De início zeramos o primeiro elemento da coluna 10 (a última coluna da matriz) com o auxílio da coluna 9. Em seguida permutamos estas duas colunas, de forma a obter:

```
 0.00   0.00   0.00   0.00   0.00   0.00   0.00   1.00   0.00   0.90
 5.00
 7.00   6.20
                3.00
                1.00  -1.33
                             -3.00
                             -1.00   2.33
-1.00   2.40   1.00   1.67   3.00   1.00   3.87
-2.00  -2.20  -3.00   2.00  -1.00   0.33  -3.55  -5.61
 1.00   1.60   2.00   0.33   3.00   3.00   1.58   1.67-11.98   3.56
-1.00   1.40  -1.00   2.33  -1.00   0.33   0.26   2.66  -3.96   1.92
```

Fig. II.4.8 - A matriz depois de eliminado o elemento 1,9

Com a eliminação de um elemento, agregamos a U, pela direita, uma nova matriz U_1^*, triangular unitária. Esta matriz possui um único elemento não nulo acima da diagonal, na posição $u^*_{9,10}$, que vale:

$$u^*_{9,10} = -3,3628$$

De forma análoga, também será necessário agregar uma matriz de permutação P_1^*, para indicar a troca das colunas 9 e 10.

2.2 Resta, ainda, eliminar um elemento da primeira linha da matriz. Para fazê-lo, vamos zerar agora o elemento da coluna 10 em função da coluna 8 e, então, permutar as colunas novamente, obtendo a matriz da figura II.4.9:

0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
5.00									
7.00	6.20								
		9.00							
		1.00	-1.33						
				-3.00					
				-1.00	2.33				
-1.00	2.40	1.00	1.67	3.00	1.00	3.87			
-2.00	-2.20	-3.00	2.00	-1.00	0.33	-3.55	1.67		-5.61
1.00	1.60	2.00	0.33	3.00	3.00	1.58	3.07	-11.98	1.67
-1.00	1.40	-1.00	2.33	-1.00	0.33	0.26	1.13	-3.96	2.66

Fig. II.4.9 - A matriz ao final do passo 2

Mais uma vez utilizamos uma matriz de permutação de colunas, P_2^*, e uma matriz triangular superior U_2^*, cujo único elemento não nulo acima da diagonal é:

$$u^{*}_{8,10} = -0,2972$$

3 Uma vez obtida a matriz $L^{*} = \begin{bmatrix} 1.e^{T}_{10} \\ \tilde{L} \end{bmatrix}$. Podemos somar $v.1.e^{T}_{10}$ a $\tilde{L}$, formando $L^{\#}$, mostrada na figura II.4.10.

5.00									
7.00	6.20								
		9.00							9.00
		1.00	-1.33						1.00
				-9.00					-1.00
				-1.00	2.33				-1.00
-1.00	2.40	1.00	1.67	9.00	1.00	9.87			9.00
-2.00	-2.20	-9.00	2.00	-1.00	0.33	-9.55	1.67		-9.61
1.00	1.60	2.00	0.33	9.00	9.00	1.58	9.07	-11.98	-1.33
-1.00	1.40	-1.00	2.33	-1.00	0.33	0.26	1.13	-9.96	2.66

Fig. II.4.10 - A matriz $L^{\#}$

4 E, para finalmente obter a nova base B' decomposta, resta apenas eliminar os elementos acima da diagonal da coluna 10 de $L^{\#}$.

Para zerar $l^{\#}_{3,10}$ (primeiro elemento não nulo de $L^{\#}_{10}$), utilizamos a coluna 3, e fazemos

$$\hat{L}_{10} = L^{\#}_{10} + \hat{u}_{3,10} L^{\#}_{3} \qquad \text{(II.4.36)}$$

onde $\hat{u}_{3,10}$ é o elemento que irá caracterizar a matriz $\hat{U}_1$ a ser agregada a U^* (obtida no passo 2).

Este processo é repetido para eliminar os elementos $l^{\#}_{4,10}, \ldots, l^{\#}_{9,10}$.

°	°	°	°	°	°
°	°	°	°	°	°
°	°	°	°	°	°
°	°	°	°	°	°
-1.00	°	°	°	°	°
-1.00	-0.67	°	°	°	°
2.00	1.00	1.29	°	°	°
0.61	0.94	1.04	2.21	°	°
-9.99	-4.99	9.48	2.95	-1.12	°
9.66	9.99	4.09	4.00	2.49	2.87

Fig. II.4.11 - A coluna $\hat{L}_{10}$ depois de eliminados os elementos das linhas 9, 5, 6, 7, 8 e 9, respectivamente

Na figura II.4.11 é apresentado um resumo de todas as operações executadas neste passo. Em cada quadro é mostrada a coluna $\hat{L}_{10}$ depois de eliminado um de seus elementos. Neste exemplo, as matrizes $\hat{U}_i$ e $\hat{P}_i$, criadas a cada eliminação $\underline{i}$ são dadas por

$$\hat{P}_i = I \quad \text{(não foi preciso fazer permutações)}$$

$$\hat{U}_i = I + e_j.\hat{u}_{j,10}.e_{10}^T \qquad \text{(II.4.37)}$$

onde

$j = 3, 5, 6, 7, 8, 9;$

$\hat{u}_{3,10} = -1.000;$ $\hat{u}_{5,10} = -0.333;$ $\hat{u}_{6,10} = 0.286;$

$\hat{u}_{7,10} = -0.332;$ $\hat{u}_{8,10} = -1.328;$ $\hat{u}_{9,10} = 0.094;$

O resultado final da atualização é a obtenção de uma nova base B' dada por

$$B' = \hat{L}.\hat{U}_6.\hat{P}_6. \ \ldots \ .\hat{U}_1.\hat{P}_1.U_2^*.P_2^*.U_1^*.P_1^*.U \qquad \text{(II.4.38)}$$

onde $\hat{L}$ é a matriz exibida na figura II.4.12.

5.00									
7.00	6.20								
		9.00							
		1.00	-1.99						
				-9.00					
				-1.00	2.99				
-1.00	2.40	1.00	1.67	9.00	1.00	9.87			
-2.00	-2.20	-9.00	2.00	-1.00	0.99	-9.55	1.67		
1.00	1.60	2.00	0.99	9.00	9.00	1.58	9.07	-11.98	
-1.00	1.40	-1.00	2.99	-1.00	0.99	0.26	1.19	-9.96	2.87

Fig. II.4.12 - A matriz $\hat{L}$

Fazendo uma análise final, pode-se pensar que é desnecessário um processo tão complexo de atualização da base, já que uma opção natural seria decompor novamente esta matriz a cada ciclo do simplex.

Mas a atualização acima descrita possui a vantagem de que seu uso acelera bastante o tempo relativo à mudança de base.

Por outro lado, o uso contínuo da atualização pode provocar:

1 Um aumento indesejado de memória, uma vez que a cada ciclo do simplex estamos criando novos elementos não nulos nas matrizes L e U;

2 Um aumento dos erros na resolução dos sistemas; e

3 Até mesmo prejuízos no que concerne ao tempo de processamento do programa, em decorrencia do aumento das matrizes;

Sendo assim, toda vez que um desses problemas puder tornar o algoritmo menos eficiente, uma nova fatoração da base é executada.

SEÇÃO 5 A SOLUÇÃO DE SISTEMAS

Dadas as matrizes L e U atualizadas, vejamos agora como resolver os sistemas dos passos 1, 2 e 4 do ciclo do simplex:

Problemas do tipo B.z = y

Se queremos encontrar $\underline{z}$, vetor solução do sistema

$$B.z = y \qquad \text{(II.5.1)}$$

e dispomos da matriz B decomposta na forma

$$B.P = L.U \qquad \text{(II.5.2)}$$

então podemos escrever

$$L.U.P^{-1}.z = y. \qquad \text{(II.5.3)}$$

Assim, resolvemos, em primeiro lugar,

$$L.x = y \qquad \text{(II.5.4)}$$

depois encontramos $\underline{w}$ tal que

$$U.w = x \qquad \text{(II.5.5)}$$

e enfim obtemos $\underline{z}$ a partir de

$$P^{-1}.z = w \qquad \text{(II.5.6)}$$

2 Problemas do tipo $B^T.z = y$

Se o sistema a ser resolvido tem a forma

$$B^T.z = y \qquad \text{(II.5.7)}$$

e, da mesma forma, sabemos que

$$B.P = L.U$$

então nosso problema pode ser descrito pela equação

$$P.U^T.L^T.z = y \qquad \text{(II.5.8)}$$

E para resolvê-lo, começamos por aplicar as permutações

$$x = P^T.y \qquad \text{(II.5.9)}$$

depois encontramos $\underline{w}$ dado pelo sistema

$$U^T.w = x \qquad \text{(II.5.10)}$$

e enfim $\underline{z}$ é obtido resolvendo

$$L^T.z = w \qquad \text{(II.5.11)}$$

CAPÍTULO TERCEIRO

IMPLEMENTAÇÃO COMPUTACIONAL

SEÇÃO 1 CARACTERÍSTICAS GERAIS DO PROGRAMA

Uma vez descrito o algoritmo, chegou a hora de comentar minuciosamente o programa computacional elaborado para resolver problemas bloco-angulares.

Comecemos, então, por algumas de suas características mais gerais de implementação:

1 O programa foi elaborado com vistas a sua implantação em microcomputadores;

2 Só se utiliza a memória rápida do computador, privilegiando o tempo de execução em detrimento da capacidade de armazenar matrizes maiores. Isto porque:

a) O trabalho de armazenar e recuperar informações de disco para poder resolver problemas muito grandes tornaria o programa de tal forma lento que seria proibitivo o seu uso;

b) Os recentes avanços no desenvolvimento de microcomputadores têm ampliado consideravelmente a disponibilidade de memória de rápido acesso, elevando seu limite para além do que seria necessário para resolver a grande maioria dos problemas bloco-angulares existentes atualmente;

3 O programa foi escrito em Pascal, recomendando-se, inclusive, que o leitor deste capítulo esteja habituado com a linguagem e sua forma de representar, por exemplo, apontadores e listas ligadas. Não sendo este o caso, espera-se que uma consulta ao apêndice A seja suficiente para esclarecer todas as dúvidas que possam eventualmente surgir.

Vejamos agora como são armazenadas as matrizes e os principais vetores auxiliares relacionados ao problema que procuramos resolver e ao nosso programa, começando pela matriz tecnológica A.

Matriz tecnológica

Uma vez que a matriz A (fig. I.1.1) pode ser dividida em blocos distintos, optamos por armazená-la da seguinte forma:

1 Cada bloco A^i da matriz tecnológica do problema na forma mista[1], é armazenado num vetor real *a[i]*, seqüencialmente e por ordem de linhas.

2 O número de colunas e linhas de cada um desses blocos pode ser obtido a partir dos vetores *cest* e *ntin*, onde:

---

[1]em que são consideradas apenas as colunas que denominaremos "originais", ou seja, que não correspondem a variáveis de folga, excesso e artificiais.

a) *cest[i]* representa a soma do número de colunas dos blocos de A até o bloco *i*, inclusive. Ou seja, cada bloco A^i possui *cest[i]-cest[i-1]* colunas (*cest[0]* = *0*);

b) *nlin[i]* representa o número de linhas de A até o bloco *i*. Portanto, este bloco *i* terá *nlin[i]-nlin[i-1]* linhas.

Com isto, o elemento a^i_{jk} estará guardado na posição *a[i]^[(j-1)*(cest[i]-cest[i-1])+k]*.

Como exemplo, observa-se na figura III.2.1 que o elemento $a^2_{5,4} = 1$ está armazenado na posição *a[2]^[(5-1)*(9-4)+4]* = *a[2]^[24]*, uma vez que, na matriz mostrada na figura, *cest[1]* = 4 e *cest[2]* = 4 + 5 = 9.

3 As linhas de acoplamento da matriz são guardadas separadamente no vetor real *aest*. Aqui, o elemento da j-ésima linha e da k-ésima coluna de estoque será fornecido por *aest[j]^[k]*.

4 Para economizar memória, as colunas correspondentes às variáveis de folga (ou excesso) e artificiais de todos os blocos e do acoplamento são armazenadas seqüencialmente num único vetor inteiro *afart*, que indica em que restrição (linha) do bloco aparece cada variável e qual o seu tipo (números positivos correspondem a variáveis de folga ou artificiais e números negativos a variáveis de excesso).

5 Por conveniência do programa, estas colunas são ordenadas de forma que as primeiras correspondem sempre às variáveis de folga das restrições canalizadas, seguindo-se as demais variáveis de folga e excesso, e terminando pelas artificiais.

6 Novamente aqui, é preciso, para localizar corretamente um elemento em *afart*, lançar mão do auxílio de quatro outros vetores: *ncan*, *nfol*, *nart* e *gcol*. Assim, se considerarmos agora a matriz A do problema na forma padrão (incluindo as colunas das variáveis citadas no item 4 acima), teremos:

a) *ncan[i]* indica qual coluna de A corresponde à primeira variável de folga de uma restrição canalizada do bloco *i*;

b) *nfol[i]* indica a coluna de A relativa à primeira variável de folga (ou excesso) das demais restrições (não canalizadas) do bloco *i*;

c) *nart[i]* indica a coluna da primeira variável artificial do bloco *i* de A; e

d) *gcol[i]* fornece o índice da última coluna do bloco *i*.

Dados todos estes vetores, é possível conhecer, para cada bloco *i* da matriz A do problema na forma padrão:

i) O número total de colunas de *i*: *gcol[i]-gcol[i-1]*;

ii) O número de restrições canalizadas de *i*: *nfol[i]-ncan[i-1]*;

iii) O número de colunas correspondentes a variáveis de folga ou excesso (incluindo as do item acima): *nart[i]-ncan[i]*;

iv) O número de variáveis artificiais: *gcol[i]-nart[i]*;

v) A posição, no vetor *afart*, da primeira coluna de folga referente a uma restrição canalizada: *afart[ncan[i]-cest[i]]*, dado que o vetor *afart* não armazena as colunas originais do problema;

vi) A posição, em *afart*, da primeira coluna de folga (ou excesso) relativa a uma restrição não canalizada: *afart[nfol[i]-cest[i]]*;

vii) A posição da primeira coluna correspondente a uma variável artificial: *afart[nart[i]-cest[i]]*.

Podemos agora, então, compreender melhor o vetor *afart*, observando que se *afart[nfol[i]-cest[i]+j]* é igual a -5, então a coluna $A^{i}_{nfol[i]+j}$ da matriz A corresponde à variável de excesso (uma vez que o termo é negativo) que é necessária para tornar uma igualdade a quinta restrição [indicada por abs(-5)] do bloco *i* do problema.

Deseja-se que a figura III.2.1 contribua para se possa ter uma visão mais clara de como estas colunas são armazenadas.

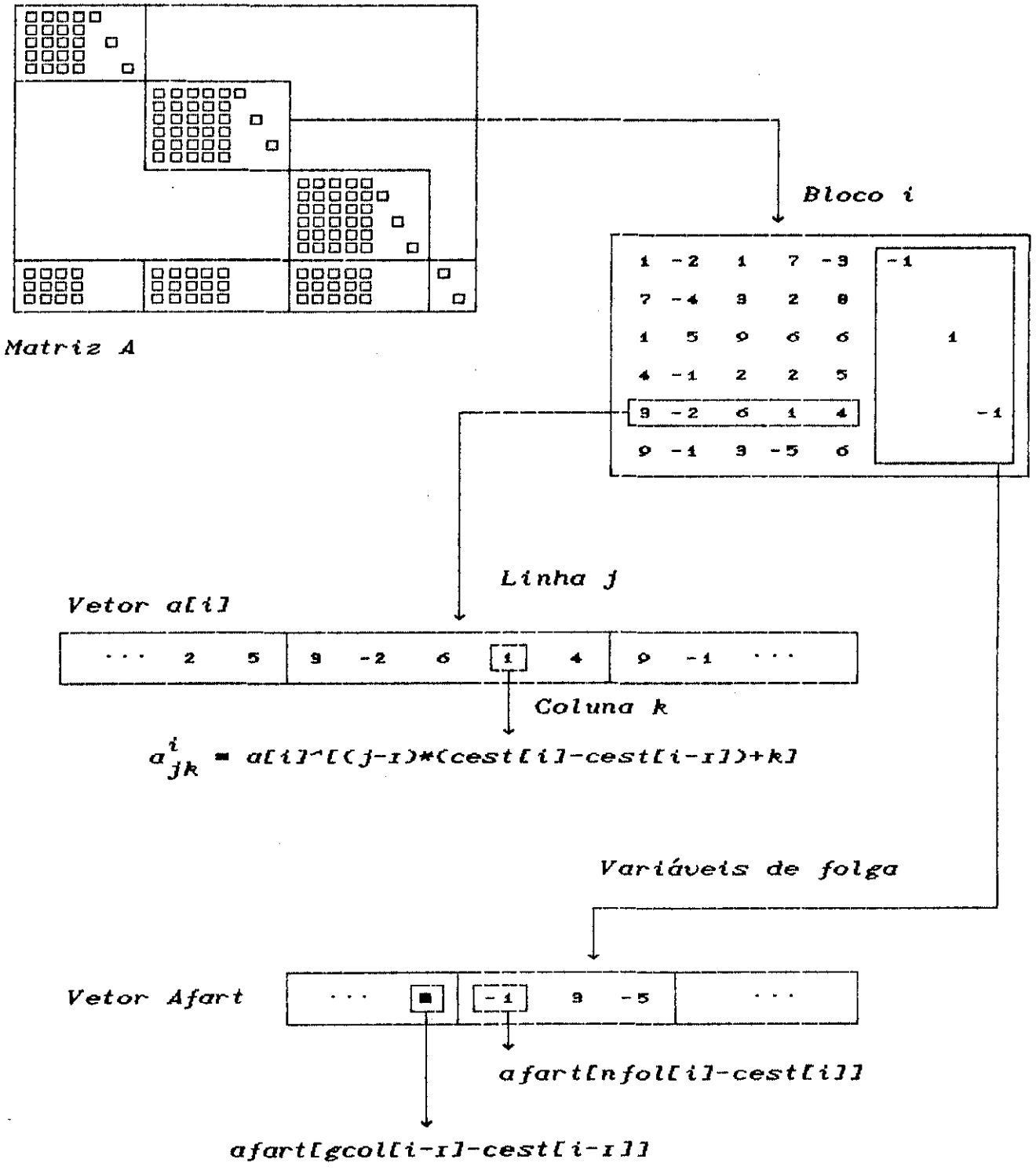


Fig III.2.1 - Armazenando os elementos dos blocos da matriz A

Estrutura de dados para a fatoração

Em virtude de termos uma base (no método simplex) com uma forma tão peculiar (densa por blocos) podemos empregar uma estrutura de dados muito simples para armazenar as submatrizes L e U originárias da fatoração.

Assim, dados os vetores inteiros *nlin* (já descrito) e *ncol* (*ncol[i]* fornece o índice da última coluna do bloco *i* da base, donde este bloco possui *ncol[i]-ncol[i-1]* colunas), temos:

Matriz triangular inferior L

l A parte da matriz L correspondente aos blocos é armazenada no vetor *l*, por colunas. A localização de um dado elemento de L é feita com o auxílio do vetor *lpos* que indica a posição, em *l*, do primeiro elemento de cada coluna dessa matriz. Assim, a coluna k de L começa na posição *l[lpos[k]]*, e o elemento l_{jk} está armazenado na posição *l[lpos[k]+j-k]*.

s Os elementos de L correspondentes às linhas de estoque são guardados em separado e por colunas. Para facilitar a atualização, conforme veremos mais adiante, os elementos da coluna L_k que estão abaixo do bloco ao qual esta coluna pertence (situado junto a diagonal da matriz) são guardados numa lista

ligada de registros que contém três campos: *dad*, *blo* e *pro*. A posição do primeiro registro da lista ligada da coluna *k* é dada por um apontador denominado *lac[k]*. Inicialmente, esta lista contém apenas informações relativas às linhas de acoplamento. O registro para o qual *lac[k]* aponta contém:

a) *lac[k]^.dad*, que fornece, inicialmente, os valores dos elementos do acoplamento da coluna k de L.

Assim, se ao término da fatoração, *lac[2]^.dad[3]* = 2,5, então o elemento $l_{tlin+3,2}$ de L vale 2,5 (*tlin* fornece o número de linhas dos blocos da base e também da matriz A);

b) *lac[k]^.blo*, que indica a que bloco os elementos de *lac[k]^.dad* pertencem.

No caso do acoplamento, *lac[k]^.blo* = *nbloc+1* (a variável *nbloc* armazena o número de blocos da matriz tecnológica A). Com isso, poderíamos dizer, de forma mais apropriada, que *lac[2]^.dad[3]* representa o elemento da coluna 2 e da linha *nlin[lac[2]^.blo-1]+3* de L (note que *nlin[nbloc]* = *tlin*).

c) *lac[k]^.pro*, que aponta para o próximo registro que contém elementos da coluna k.

Para os elementos do acoplamento, *lac[k]^.pro* = *pnil*, onde *pnil* é um registro sem valor que indica o fim da lista ligada.

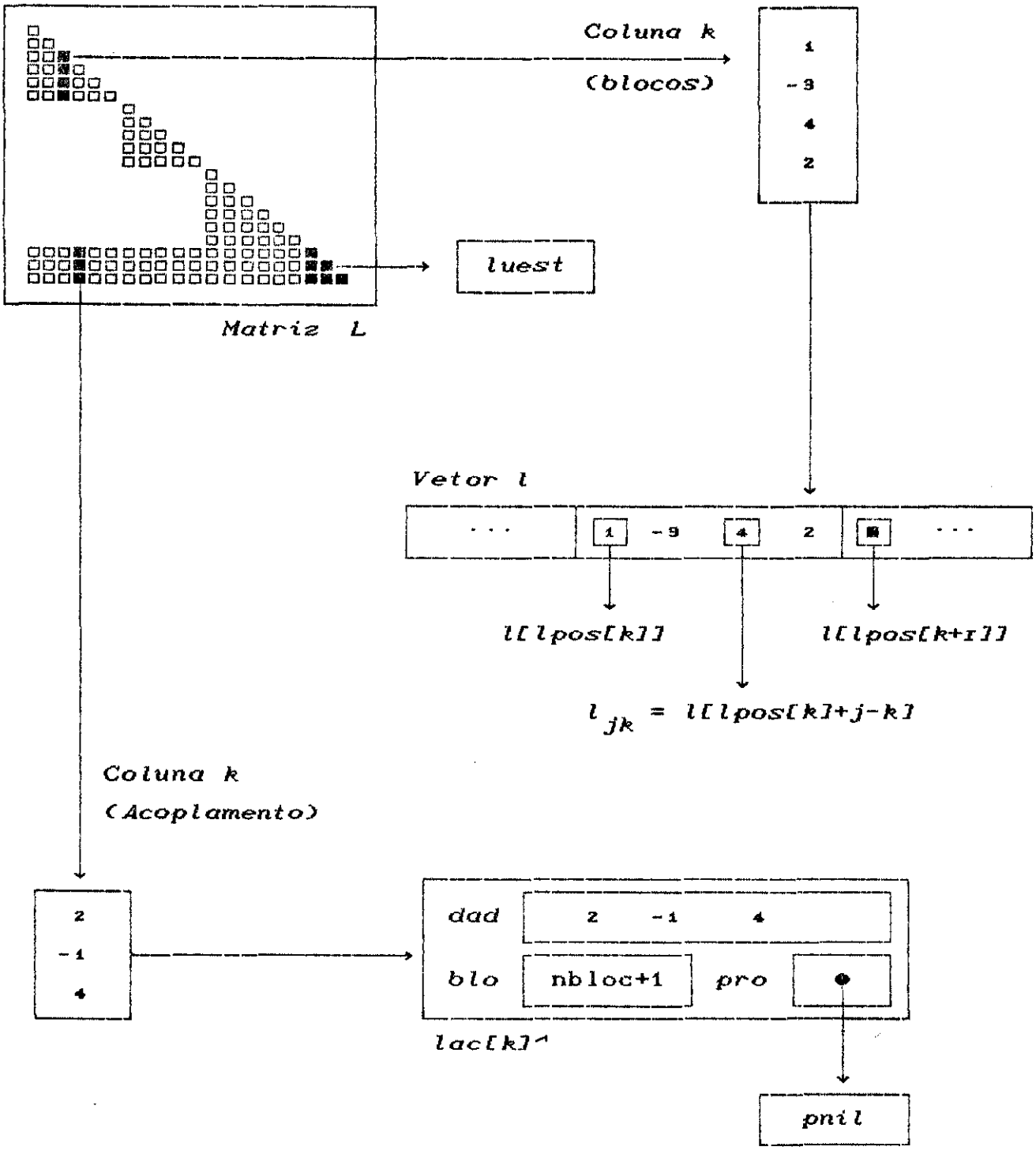


Fig. III.2.2 - Armazenando a matriz L oriunda da decomposição LU

Matriz triangular superior U

1 U, a menos das linhas do acoplamento, também é armazenada num único vetor *u*, seqüencialmente por ordem de linhas e com todos os elementos já permutados.

2 Não são necessários vetores auxiliares como os de L, dado que o vetor *u* não será modificado na atualização. Apenas um vetor *icn* indica as permutações sofridas pelas colunas que sobraram na fatoração dos blocos não quadrados de B e que foram transladadas para a direita.

Matriz de permutações P

P é armazenada num vetor *cpos*, de forma que se *cpos[i]* = *j* então a coluna que, ao final da fatoração, está na posição *i* de L e U é a coluna *j* original da base.

Alguns elementos das restrições de estoque

Os elementos das restrições de estoque que não estão sob os blocos de L são guardados na matriz *luest* (vide figura III.2.2).

Assim, o elemento *luestli,jl* pertence a i-ésima linha de estoque e se refere à coluna *j+tlin* da base. Este elemento fará parte de U se i<j, caso contrário pertencerá a L.

Outras estruturas de dados estudadas para a fatoração

Acredita-se que a forma de armazenamento recém descrita seja a mais apropriada para o tipo de problema com que nos defrontamos. Tal crédito está fundamentado no fato de ter sido essa a estrutura de melhor desempenho dentre todas as alternativas testadas para guardar as matrizes oriundas da decomposição da base. No rol das opções descartadas podemos incluir, por exemplo, aquelas que envolvem:

r Armazenar L e U exatamente como a matriz A, ou seja, reunidas bloco a bloco e com os elementos dispostos seqüencialmente por ordem de linhas, fazendo com que os que estiverem acima da diagonal principal pertençam a U e os demais a L.

Esta seria, talvez, a melhor alternativa se fossemos fatorar a matriz a cada passo do simplex, mas como implementamos um processo mais rápido de atualização da base, ela se mostrou um pouco ineficiente do ponto de vista global;

2 Copiar os elementos de A no vetor que irá armazenar L e U e depois atualizar estes valores;

Uma variante que também pode ser considerada ligeiramente menos eficiente;

3 Armazenar U como um vetor de triplas (u,x,y), onde x e y são as coordenadas de cada um dos elementos de U.

É a opção correta quando os elementos de U não estão em posições conhecidas. Mas, uma vez que U tem uma estrutura bastante particular que não se altera durante a fatoração, esta alternativa induz um aumento desnecessário de memória, sem acelerar o processo de resolução de sistemas;

4 Armazenar os vetores auxiliares por bloco, e não em vetores únicos para toda a matriz;

Opção descartada por ser pouquíssimo mais lenta que a adotada;

5 Não permutar diretamente os elementos de U durante a fatoração, mantendo-se, em contrapartida, um vetor que torne possível a identificação da posição de cada um destes elementos toda vez que for necessário resolver algum sistema.

Indicada quando é necessário resolver um número pequeno de sistemas com L e U, esta alternativa também não foi

adotada em virtude de estarmos interessados em implementar algum processo rápido de atualização da base;

6 Armazenar por linhas todos os elementos relativos ao acoplamento das matrizes L e U;

Esta variante é ligeiramente mais econômica em termos de memória do que aquela adotada. É, inclusive, utilizada no algoritmo acelerado que será apresentado no capítulo seguinte. Mas, nesta primeira implementação, também não foi adotada pois tornava mais lento o atual processo de atualização.

Estrutura de dados para a atualização

Para a atualização adotaremos a seguinte estrutura de dados:

Matriz U

1 A matriz U obtida na fatoração permanece intacta. Digamos que ela ocupe as primeiras *ufimf* posições do vetor *u*. A atualização então é feita anexando, a partir da posição *u[ufimf+1]*, os elementos acima da diagonal das matrizes

triangulares superiores unitárias, U_k, criadas na atualização (cada uma dessas matrizes só possui um elemento não nulo acima da diagonal).

e A linha e a coluna de cada um desses elementos, além da permutação executada antes de se aplicar U_k, são guardados no vetor *upos* que possui registros com três campos distintos: *x*, *y* e *p*.

Assim, se em dado momento permutamos as colunas *i* e *j* e, em seguida, aplicamos U_k cujo único elemento nao nulo é $u^k_{i,j} = v$, então teremos:

a) *ulufimf+k]* = *v*

b) *upos[k].x* = i (linha do elemento *v*)

c) *upos[k].y* = j (coluna do elemento *v*)

d) *upos[k].p* = i (se *upos[k].p* ≠ *upos[k].y* então as colunas dadas por estas duas variáveis foram permutadas. No caso delas seram iguais, não houve permutação).

Matriz L

I Para a matriz L, além dos dados guardados seqüencialmente no vetor *l*, que são atualizados quando necessário, optou-se por armazenar os novos elementos não nulos

oriundos das atualizações numa estrutura mista que é em parte estática, mas que também emprega listas ligadas.

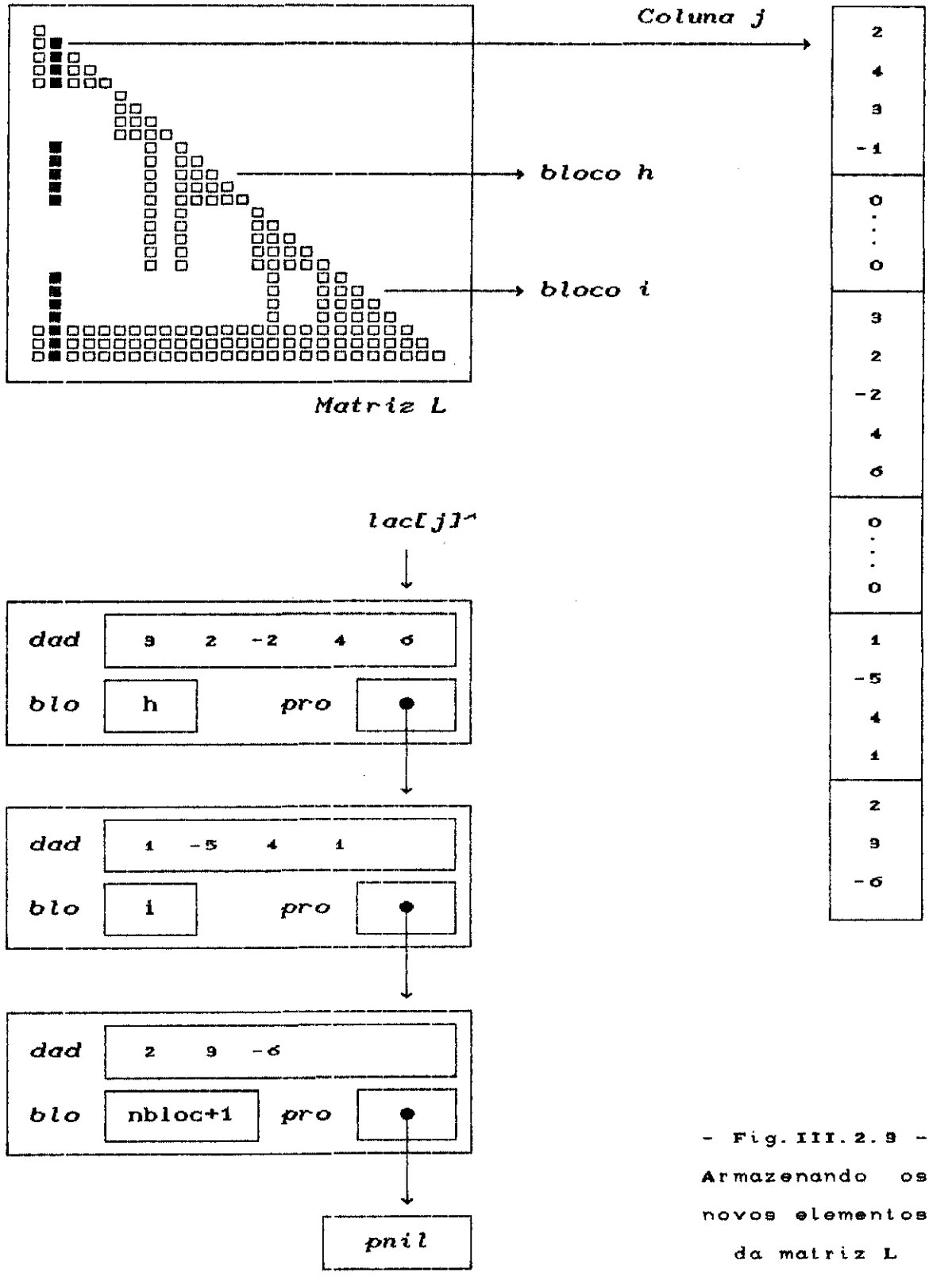
2 Assim, se a uma coluna j foram agregados elementos à altura do bloco i (obviamente $j \leq gcol[i-1]$), então é criado um registro t que será ligado à lista ligada que se inicia em $lac[j]$, de forma que

a) $t^{\wedge}.blo$ = i;

b) $t^{\wedge}.dad$ armazena os novos elementos acrescentados a L;

c) $t^{\wedge}.pro$ aponta para o próximo registro da lista.

Na figura III.2.3 temos uma noção visual da estrutura de L para a atualização.



- Fig. III.2.9 - Armazenando os novos elementos da matriz L

Outra opção de armazenamento dos novos elementos de L

Uma opção descartada (embora testada) para guardar os elementos surgidos durante a atualização consiste em empregar uma estrutura puramente estática para armazenar L.

Neste caso, são deixadas algumas posições vagas entre as colunas no vetor *l* (vide fig. III.2.4), posições estas que são utilizadas para armazenar os elementos não nulos eventualmente criados na atualização.

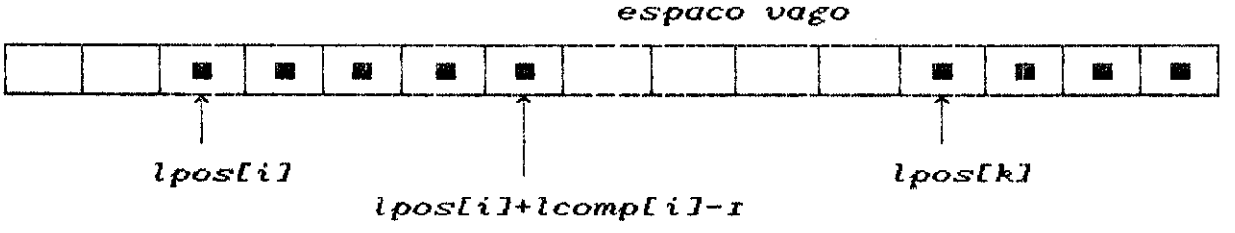


Fig. III.2.4 - Outra estrutura de dados para armazenar L

Se o acréscimo à coluna i de L não couber no espaço vago que a sucede em *l*, toda a coluna é removida para o final deste vetor.

Quando o vetor *l*, após diversas atualizações, torna-se demasiadamente grande, seus elementos são comprimidos, eliminando os espaços que se tornaram vagos quando da remoção de colunas.

Uma nova fatoração passa a ser executada na hipótese de não haver mais espaço disponível em *l* para armazenar novos elementos ou quando estes estiverem em tal número, que venha a tornar ineficiente, do ponto de vista do tempo, a resolução dos sistemas II.5.1 e II.5.7.

Esta estrutura não exige que as colunas de L sejam armazenadas em ordem, mas apenas que seus elementos estejam em posições contíguas do vetor *l*.

Mas, por outro lado, é preciso manter mais um vetor, *lcomp*, que indique qual o comprimento atual de cada coluna em *l*.

Os motivos desta alternativa ter sido abandonada são:

1 Em cada coluna L_k de L, os elementos originarios da atualização são criados geralmente em blocos que muitas vezes estão distantes uns dos outros;

2 O elementos de L_k são requisitados seqüencialmente na atualização e na resolução de sistemas, o que torna eficiente o uso de listas ligadas.

Exemplo numérico completo

Vamos aproveitar o problema mostrado na figura III.2.5 para permitir que se tenha uma visão global da estrutura de dados descrita nesta seção.

minimizar

$$-2x_1 \; -1x_2 \; -1x_3 \; -4x_4 \; -7x_5 \; -1x_6 \; -3x_7 \; -1x_8 \; -6x_9 \; -2x_{10} \; -5x_{11}$$

sujeito a

$$\begin{array}{rrrrrrrrrrrcr}
1x_1 & +2x_2 & +5x_3 & -3x_4 & & & & & & & & \leq & 8 \\
2x_1 & -1x_2 & +7x_3 & +2x_4 & & & & & & & & \leq & 12 \\
 & & & & +1x_5 & +3x_6 & +3x_7 & & & & & \geq & 10 \\
 & & & & -1x_5 & +1x_6 & +1x_7 & & & & & = & 2 \\
 & & & & & & & +1x_8 & -1x_9 & -3x_{10} & +2x_{11} & \leq & 1 \\
 & & & & & & & +1x_8 & +1x_9 & -1x_{10} & +3x_{11} & \geq & 7 \\
-1x_1 & +2x_2 & -1x_3 & +3x_4 & +2x_5 & +1x_6 & +1x_7 & -2x_8 & +1x_9 & +3x_{10} & -1x_{11} & \leq & 9 \\
2x_1 & -3x_2 & -2x_3 & -1x_4 & +1x_5 & -3x_6 & -2x_7 & -4x_8 & +5x_9 & -1x_{10} & +1x_{11} & \leq & 5 \\
3x_1 & +1x_2 & +1x_3 & +1x_4 & +1x_5 & +2x_6 & -2x_7 & +1x_8 & -1x_9 & +3x_{10} & +1x_{11} & = & 10 \\
1x_1 & -1x_2 & -1x_3 & +2x_4 & +2x_5 & -1x_6 & +3x_7 & +3x_8 & -1x_9 & -1x_{10} & +1x_{11} & \geq & 14
\end{array}$$

e

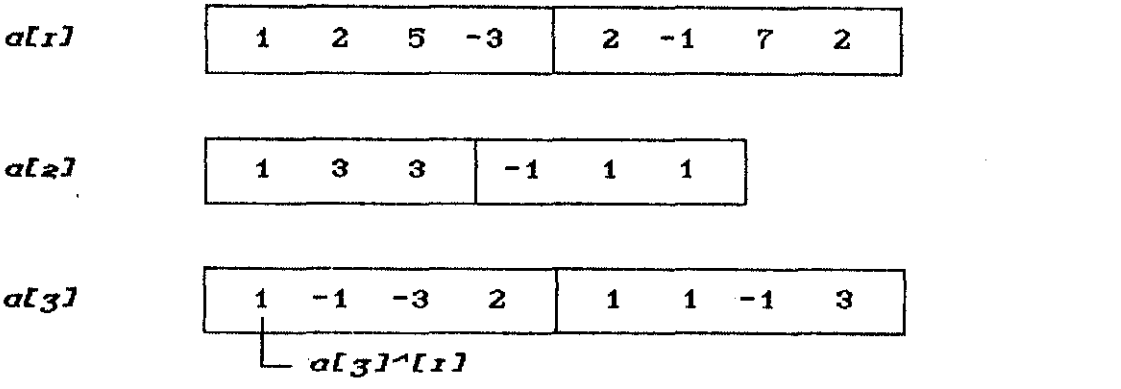
$$x_1 \leq 6 \qquad x_3 \leq 8 \qquad x_5 \leq 6 \qquad x_7 \leq 7 \qquad x_9 \leq 4 \qquad x_{11} \leq 5$$

$$x_2 \leq 4 \qquad x_4 \leq 3 \qquad x_6 \leq 8 \qquad x_8 \leq 2 \qquad x_{10} \leq 9$$

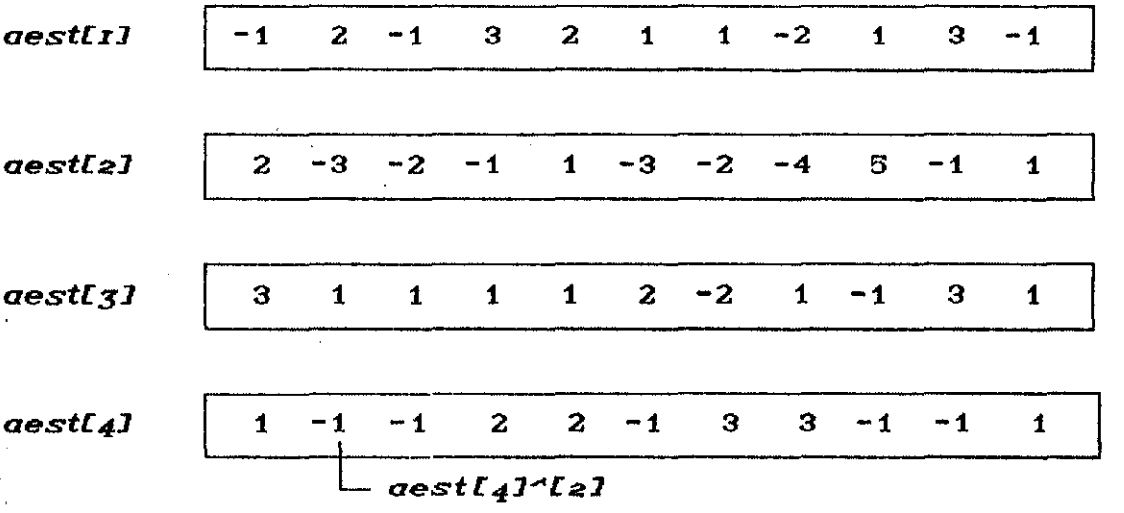
Fig. III.2.5 - Um exemplo de problema com estrutura bloco-angular

Vetores relacionados aos dados do problema

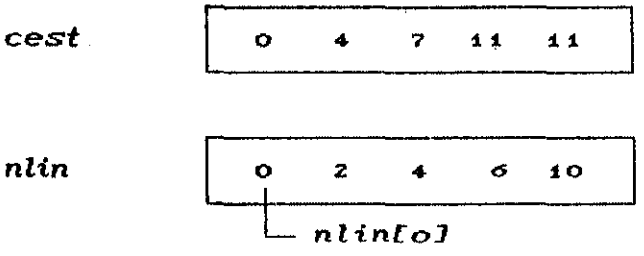
Os blocos da matriz A mostrada na figura III.2.5 são armazenados nos vetores abaixo (as divisões representam mudanças de linhas da matriz):



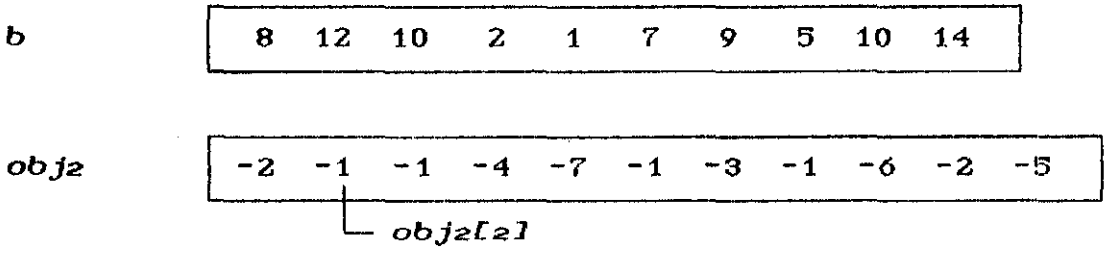
As linhas de acoplamento de A são guardadas nos vetores *aest*:



Os vetores que permitem o acesso aos dados de A são:



Os vetores b e c são guardados, respectivamente, em b e obj_2:



Algumas outras variáveis importantes são:

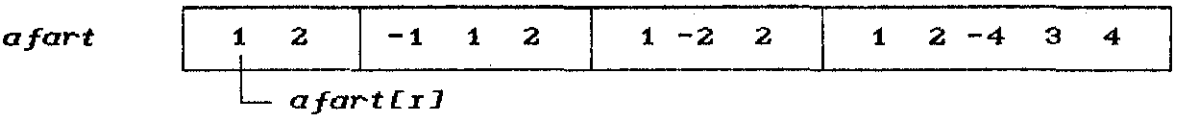
$$nbloc = 3; \qquad nlest = 4; \qquad tlin = 6;$$

Se colocarmos o problema na forma padrão, aparecerão variáveis de folga, excesso e artificiais, de acordo com quadro III.2.1, dado abaixo:

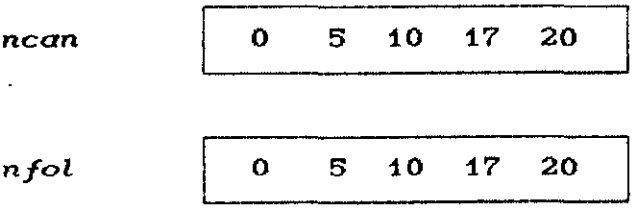
Varíaveis	Linhas
de folga	1, 2, 5, 7, 8
de excesso	3, 6, 10
artificiais	3, 4, 6, 9, 10

- Quadro III.2.1 -

A posição e o tipo de cada variável guardamos no vetor *afart*. Bloco a bloco, são armazenadas primeiro as variáveis de folga e excesso (estas com sinal negativo), depois as artificiais (as linhas separam blocos):



Uma consulta ao vetor *afart* só é possível com o auxílio dos vetores:



nart

0	7	11	19	23

gcol

0	6	12	19	24

gcol[o]

Deste último vetor, podemos concluir que o primeiro bloco de A possui 6 colunas (4 "originais" e 2 de folga). Logo, o segundo bloco irá começar pela coluna 7, e, como vemos, terminará na coluna 12, possuindo, portanto, 6 colunas também (3 "originais", 1 de excesso e 2 artificiais). O bloco 3 engloba as colunas 13 a 19, e o acoplamento as de número entre 20 e 24 (2 de folga, 1 de excesso e 2 artificiais).

Observamos também que, com a ausência de restrições canalizadas, o vetor *ncan* é idêntico a *nfol*.

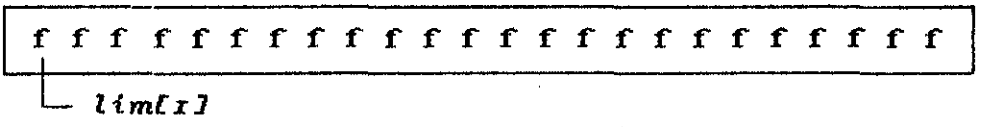
Por fim, as canalizações das variáveis são armazenadas no vetor *lsup*, considerando, inclusive, as variáveis de folga, excesso e artificiais, que não têm limite (veja que se houvessem restrições canalizadas, as variáveis de folga correspondentes teriam limite superior).

lsup

6	4	8	3	∞	∞	6	8	7	∞	∞	∞	2	4	9	5	∞	∞	∞	∞	∞	∞	∞	∞

lsup[1]

Para cada variável, indicamos se esta está no seu limite superior (da canalização) através do vetor booleano *lim*, mostrado abaixo (t representa *true* e f corresponde a *false*). Vemos que, inicialmente, todas as variáveis estão abaixo do limite.



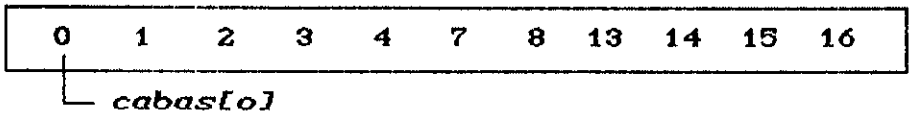
Com esta implementação, consideramos que todas as variáveis são maiores ou iguais a zero. Nos casos em que isto não ocorre pela presença de limites inferiores, o vetor *lsup* deve armazenar a diferença entre os limites.

Vetores relacionados à base

Suponhamos que as colunas da matriz A pertencentes à base sejam as de número 1, 2, 3, 4 (do bloco 1), 7 (já no bloco 2), 8, 13 (pertencente ao bloco 3), 14, 15 e 16. A base seria, então, a matriz mostrada na figura II.3.6 (pag. 51).

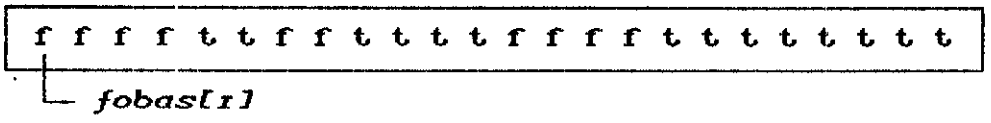
Para esta matriz, indicamos os índices das colunas que estão na base através do vetor *cabas*:

cabas



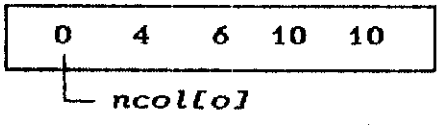
Por outro lado, indicamos no vetor *fobas* quais variáveis não estão na base (*fobas[i]*=t indica que *i* não está na base):

fobas



O número de colunas de cada bloco da base deve ser extraído do vetor *ncol*.

ncol

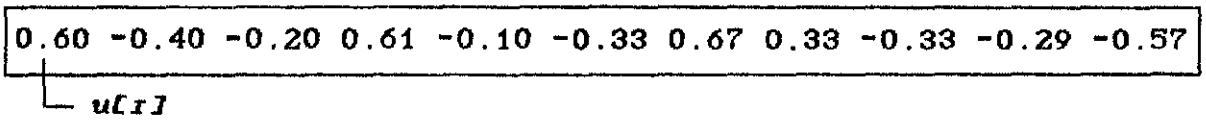


Os elementos dos blocos da base, decomposta nas matrizes L e U[2], são armazenados nos vetores:

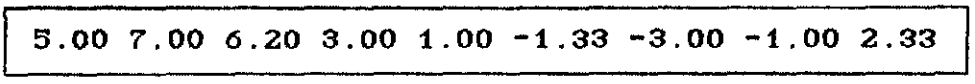
---

[2]como já vimos na figura II.3.12, à página 57.

u

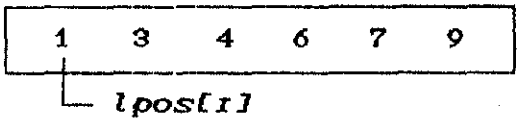


l

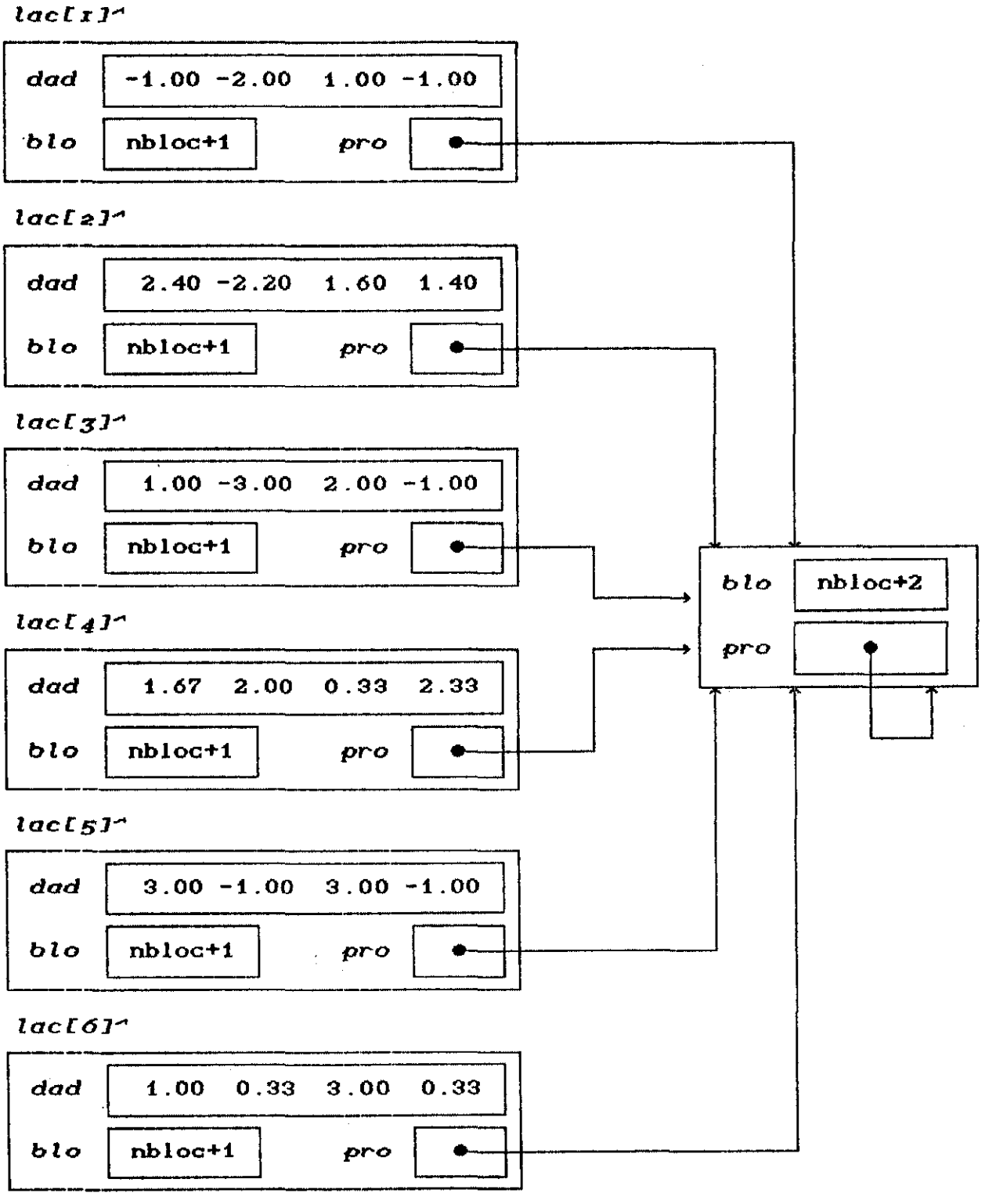


Para endereçar corretamente os elementos de *l* usamos:

lpos



Os elementos das linhas de acoplamento da base são encontrados em:


lac[1]
dad
-1.00 -2.00 1.00 -1.00
blo
nbloc+1
pro
lac[2]
dad
2.40 -2.20 1.60 1.40
blo
nbloc+1
pro
lac[3]
dad
1.00 -3.00 2.00 -1.00
blo
nbloc+1
pro
lac[4]
dad
1.67 2.00 0.33 2.33
blo
nbloc+1
pro
lac[5]
dad
3.00 -1.00 3.00 -1.00
blo
nbloc+1
pro
lac[6]
dad
1.00 0.33 3.00 0.33
blo
nbloc+1
pro
blo
nbloc+2
pro

e também na matriz *luest*:

luest

luest[r,r]			
3.87	0.33	0.27	0.15
-3.55	-5.61	0.30	0.82
1.58	1.67	3.56	0.59
0.26	2.66	1.92	2.51

A matriz de permutações *P*, mostrada na figura II.3.13 (pág. 58), é armazenada no vetor:

cpos

3	4	6	5	9	10	2	7	1	8

Vetores da base após as atualizações

Se procedermos, como no exemplo do capítulo 2, substituindo a sétima coluna da base pela terceira coluna do segundo bloco de A, obteremos a matriz L dada pela figura II.4.12 (pág. 82). Para armazenarmos esta matriz usamos os vetores *l*, *lac* e *luest* já expostos. Destes, o único que sofreu

modificações nesta atualização foi o vetor *luest*, mostrado abaixo:

luest

luest[1,1]			
3.87	0.33	0.27	0.15
-3.55	1.67	0.30	0.82
1.58	3.07	-11.98	0.59
0.26	1.13	-3.96	2.87

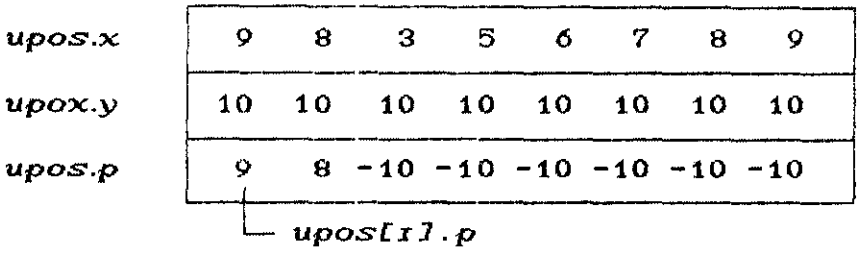
Por outro lado, ao vetor *u* foram adicionados vários elementos novos. Estes elementos são armazenados a partir da posição *ufimf+1*:

u

... -0.57	-3.36 -0.30 -1.00 -0.33 0.29 -0.33 -1.33 -0.09

u[ufimf]

Para conhecer a posição (linha e coluna) dos elementos de *u*, e as permutações ocorridas na atualização, utilizamos o vetor *upos*, lembrando que os termos positivos em *upos.p* correspondem à eliminação de elementos de ω (vide fig. II.4.2, pág. 63) e os negativos estão relacionados à eliminação de elementos de $L^{\#}$ (fig II.4.3, pág. 66).



De posse da estrutura de dados, resta ainda, ao leitor, tomar conhecimento dos principais passos do programa. Isto será feito na ordem natural com que são executadas as subrotinas que os contêm e de acordo com um critério de abrangência, ou seja, jamais abordando uma rotina sem que antes tenham sido descritas aquelas que a requisitam e que estão, portanto, em nível superior.

A figura III.3.1 apresenta todas as rotinas elaboradas, indicando de uma forma geral quem as requisita.

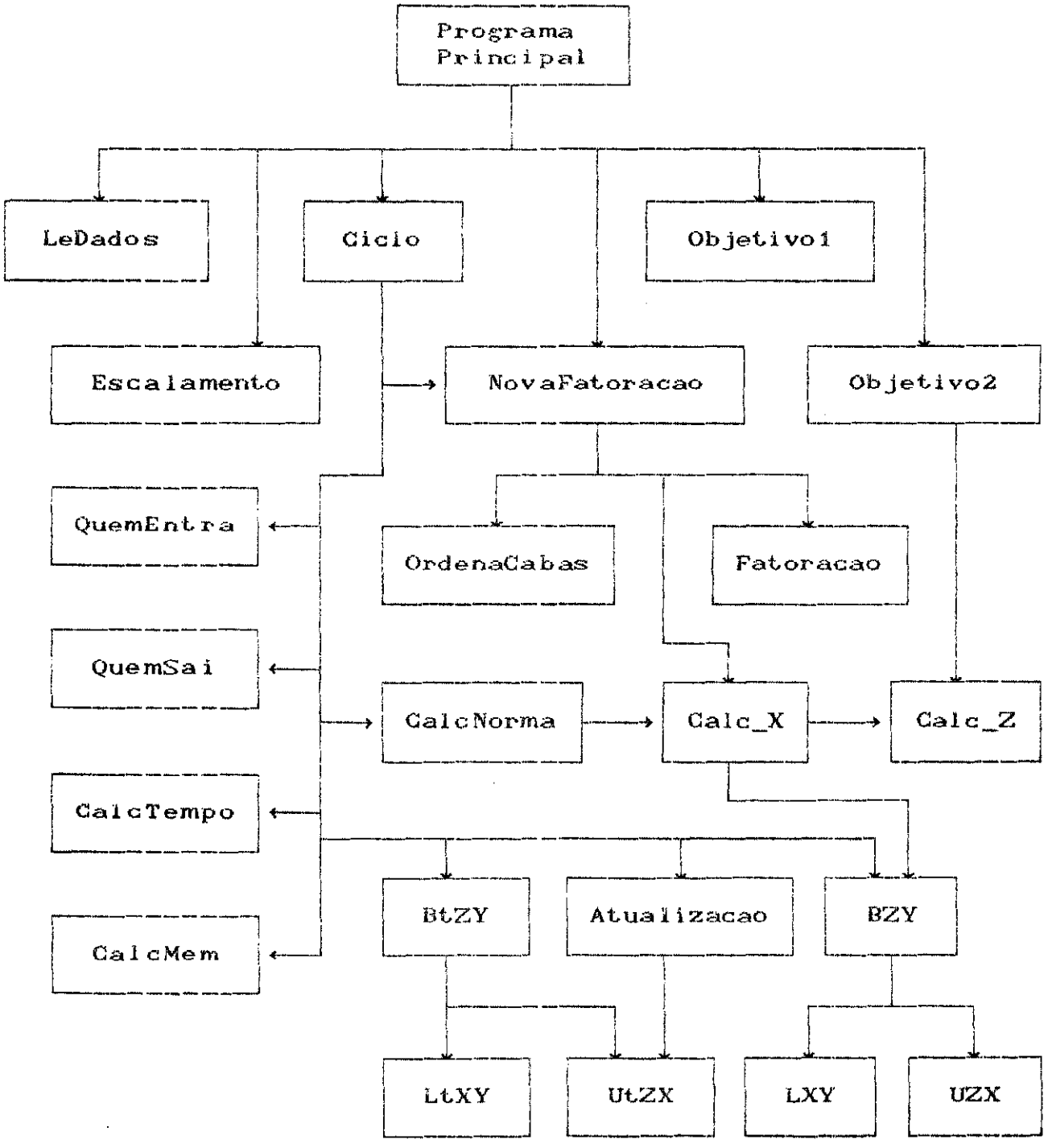


Fig. III.3.1 - Diagrama geral das rotinas do programa

E uma vez que a linguagem utilizada é o Pascal, achou-se por bem empregar, na descrição de cada procedimento, o que se chama de *pseudo-code* (pseudo-código? pseudo-programa?), e que muito se assemelha com a linguagem, mas é facilmente compreendido, sem exigir nenhum conhecimento específico.

Comecemos, então, analisando como é executada a

Rotina principal

Podemos resumir o algoritmo geral do simplex a:

1. Ler a matriz *A* e vetores *b* e *c* (dados do problema)
2. Escalar a matriz *A*
3. Criar a função objetivo da fase 1 do simplex
4. Determinar a base inicial
5. Decompor a base
6. Repetir
   1. O ciclo do simplex
7. Até que não haja candidatos a entrar na base ou que o valor da função objetivo, z, seja igual a zero

8. Se z > 0

    1. Então não há solução factivel para o problema

    2. Senão

        1. Passar a usar a função objetivo original, c

        2. Eliminar as variáveis artificiais

        3. Repetir

            1. O ciclo do simplex

        4. Até que não haja candidato a entrar ou sair da base

        5. Se há candidato a entrar na base

            1. Então a solução é ilimitada

            2. Senão a solução ótima foi encontrada

Obviamente, os passos mais importantes e complexos do algoritmo acima precisam ser melhor descritos. Isto será feito quando forem comentadas as subrotinas que a eles se referem . Os nomes das rotinas e os passos correspondentes são dados no quadro III.3.1 abaixo.

Passos	Procedimento
1	LeDados
2	Escalamento
3, 4	Objetivo1
5	NovaFatoração
6.1, 8.2.3.1	Ciclo
8.2.1	Objetivo2

- Quadro III.9.1 -

Procedimento Objetivo1

Dado que a subrotina *LeDados* é pouco importante para nosso estudo e que há neste capítulo uma seção própria sobre o escalamento, podemos começar a descrever as rotinas requisitadas pelo programa principal dado acima partindo de **Objetivo1**, cujo *pseudo-code* mostramos a seguir:

1. Para toda variável i
   1 Se esta for artificial
      1. Então seu custo é 1 ($c[i] = 1$)
      2. Senão seu custo é 0 ($c[i] = 0$)

2. Para toda variável i

   1. Se esta for artificial ou de folga

      1. Então ela está na base (*nbas[i] = true*)

      2. Senão está fora da base (*nbas[i] = false*)

3. Para toda variável i não básica

   1. O valor de x_i é igual a zero (*lim[i] = false*)

Procedimento Objetivo2

Para manter uma certa coerência, antes de passar ao ciclo do simplex propriamente dito preferimos descrever como é feita a mudança de fases no programa, executada por intermédio do procedimento **Objetivo2**:

1. Eliminar as variáveis artificiais do problema
2. Substituir a função objetivo pela original do problema, c
3. Recalcular $z = cx$

O passo 3 desta rotina é executado pelo procedimento **Calc_Z**; dado abaixo.

Procedimento Calc_Z

1. Fazer $z = 0$
2. Para toda variável i
    1. Se o seu limite inferior é diferente de 0
        1. Então somar a z este valor multiplicado por c_i [3]
3. Para toda variável não básica i
    1. Se esta encontra-se em seu limite superior
        1. Então somar a z este valor multiplicado por c_i
4. Para toda variável básica i
    1. Somar a z o termo $c_i.x_i$

NovaFatoração

Imediatamente antes de fatorar a base, é preciso executar algumas instruções relacionadas a este processo, motivo pelo qual foi criada a subrotina **NovaFatoração**, que se resume a:

---

[3]Este passo é executado apenas uma vez. O valor do somatório é, então, guardado pela variável *zinic*.

1. Substituir o índice da coluna que sai pelo da que entra na base

2. Ordenar as colunas da base segundo seus índices

3. Atualizar o número de colunas de cada bloco da base

4. Eliminar os registros acrescentados às listas ligadas das colunas de L

5. Fatorar a base B

6. Calcular a solução atual (resolver $Bx = b$ e calcular z)

7. Atualizar as variáveis relativas ao tempo de execução e à memória disponível

Os passos desta rotina que requisitam outros procedimentos são:

Passos	Procedimento
2	OrdenaCabas
5	Fatoração
6	Calc_X

- Quadro III.3.2 -

Destas subrotinas, Fatoração será objetivo de uma seção específica, a de número 4 deste capítulo, donde resta

ainda comentar sobre **OrdenaCabas** e **Calc_X**. Assim sendo:

Procedimento OrdenaCabas

Este procedimento extremamente simples consiste em:

1. Arranjar as colunas da base por ordem crescente de seus índices, usando um método de inserção (*Insertion Sort*)

Procedimento Calc_X

1. Copiar no vetor γ o vetor b
2. Para toda variável i
    1. Se esta variável está em seu limite superior então
        1. Fazer $\gamma = \gamma + A_i.x_i^{sup}$
3. Resolver $B.x = \gamma$
4. Calcular z

Este procedimento, como tantos outros, também requisita algumas rotinas, as quais encontram-se enumeradas no quadro abaixo.

Passos	Procedimento
3	BZY
4	Calc_Z

- Quadro III.3.3 -

Destas subrotinas, o procedimento **Calc_Z** já foi descrito e **BZY** pode ser encontrado mais adiante, na seção 6 deste capítulo.

Subrotina Ciclo

Finalmente chegamos ao ciclo do simplex propriamente dito, parte crucial do programa. Procuraremos, aqui, dar uma noção mais prática de como é implementado o algoritmo que já foi apresentado na seção 2 do capítulo segundo. Em resumo, temos que:

1. Incrementar o número de iterações, *niter*
2. Se *niter* ultrapassa o máximo permitido
    1. Então parar o programa e acusar fracasso

3. Encontrar os multiplicadores λ (resolver $B^T\lambda = C_B$)

4. Decidir qual variável irá entrar na base

5. Se existir ao menos uma candidata a entrar, então

    1. Resolver $B.y = A_{ent}$

    2. Decidir qual variável irá sair da base

    3. Se houver ao menos uma candidata a sair, então

        1. Se o número de atualizações consecutivas tiver excedido o máximo ou se $\|b-Ax\|$ for maior que o tolerável ou o tempo da última atualização tiver sido muito grande ou a memória disponível for insuficiente para uma atualização

            1. Então fazer uma nova fatoração da base

            2. Senão atualizar a base

A subrotina **Ciclo** requisita várias outras que, como de hábito, estão resumidas no quadro III.3.4:

Passos	Procedimento
3	BtZY
4	QuemEntra
5.1	BZY
5.2	QuemSai
5.3.1	CalcNorma
5.3.1	CalcTempo
5.3.1	CalcMem
5.3.1.1	NovaFatoração
5.3.1.2	Atualização

- Quadro III.3.4 -

A seguir descreveremos as subrotinas **QuemEntra**, **QuemSai**, **CalcNorma**, **CalcTempo** e **CalcMem**. As demais serão comentadas nas próximas seções.

Procedimento QuemEntra

Esta é a rotina que executa o teste de otimalidade do simplex.

A despeito de outras opções eventualmente mais interessantes, optou-se por implementar o critério de Dantzig para a seleção da coluna que entra na base, de forma que se possa ter uma comparação honesta com os outros algoritmos descritos na seção 3 do capítulo primeiro, pois todos adotam esta estratégia.

Assim, devemos:

1. Fazer μ_{min} (menor custo reduzido encontrado) igual a 0
2. Para cada coluna i de A
    1. Se esta coluna não é básica, então

1. Calcular $\mu_i = c_i - \lambda.A_i$
2. Se $\mu_i < \mu_{min}$ e x_i estiver no limite inferior
    1. Então esta será a coluna que entrará na base
    2. Senão se $\mu_i > -\mu_{min}$ e x_i estiver no limite superior
        1. Então esta será a coluna que entrará na base

Procedimento QuemSai

Apresentaremos agora a rotina que executa o teste da razão para a escolha da componente que sai da base, a qual sugestivamente denominamos **QuemSai**, e que consiste em:

1. Atribuir ∞ a ν_{sai} (menor valor obtido no teste da razão)
2. Se a variável que entra na base, x_{ent}, vale 0 atualmente
    1. Então $\delta = 1$
    2. Senão $\delta = -1$
3. Para toda componente i da base
    1. Se $y_i \neq 0$ então
        1. Se $\delta.y_i > 0$

1. Então se $\frac{x_i}{|y_i|}$

    1. Então a variável i deverá deixar a base

2. Senão se $\frac{x_i^{sup} - x_i}{|y_i|}$

    1. Então a variável i deverá deixar a base

4. Se o limite superior da variável que deixa a base for menor que ν_{sai}, então

    1. Mudar o limite da candidata a entrar na base (do limite inferior para o superior ou vice-versa)

    2. Fazer ν_{sai} igual ao limite da variável que entra na base

5. Para toda variável x_i pertencente à base

    1. Fazer $x_i = x_i - \delta.\nu_{sai}$

6. Se existe uma candidata, k, a sair da base

    1. Atribuir a esta variável seu limite mais próximo

    2. Fazer $x_{sai} = x_{sai} + \delta.\nu_{sai}$

Função CalcNorma

Vejamos como calcular $\|b - Ax\|_\infty$ e verificar se este valor é menor que o permitido:

1. Atribuir zero à variável *norma*
2. Para toda componente *i* da base
    1. Calcular $\left| b_i - \sum_j A_{ij} x_j \right|$
    2. Se este termo for maior que *norma*
        1. Então atribuir a *norma* tal valor
3. Se *norma* for maior que uma tolerância
    1. Então *CalcNorma* = true
    2. Senão *CalcNorma* = false

Função CalcTempo

Para determinar se vale a pena, sob o ponto de vista do tempo consumido pelo programa, refatorar a base numa determinada iteração do simplex, definimos um fator de economia (que será devidamente descrito ao final do capítulo), que é menor que a unidade nos casos em que a atualização é fortemente recomendada.

A decisão acerca da conveniência de uma nova fatoração da base é tomada por esta função, que se resume a

1. Calcular o fator de economia de tempo
2. Se este fator for menor que 0.999
    1. Então
        1. *CalcTempo* = false
    2. Senão se acabamos de executar *CalcNorma* ou se o fator é menor que o da iteração anterior
        1. Então *CalcTempo* = false, (pode ser feita uma atualização)
        2. Senão *CalcTempo* = true, (é preciso refatorar a base)

Função CalcMem

Finalmente, para verificar se há memória suficiente para uma nova atualização, precisamos:

1. Verificar a disponibilidade de memória
2. Se o espaço existente for suficiente para uma atualização
    1. Então *CalcMem* = false
    2. Senão *CalcMem* = true

Nesta seção veremos como é feita a decomposição da base nas matrizes triangulares L e U. O procedimento que executa este passo chama-se **Fatoração**, e podemos decrevê-lo com o auxílio do *pseudo-code* abaixo.

1. Para todo bloco h entre 1 e *nbloc*
    1. Indicar no vetor *cpos* que não foi feita ainda nenhuma permutação
    2. Para toda linha i do bloco h da base
        1. Copiar a linha i da submatriz restante de B para o vetor U, levando em consideração as permutações efetuadas nas iterações anteriores
        2. Aplicar nesta linha os fatores das linhas superiores
        3. Escolher como pivô o elemento de maior valor absoluto da linha (que diremos estar na coluna *pivo*)
        4. Se o pivô for igual a zero
            1. Então parar o programa (a base é singular)
        5. Permutar os elementos das colunas i e *pivo* de U
        6. Permutar também *cpos[i]* e *cpos[pivo]*
        7. Guardar o elemento pivô na posição l_{ii}

8. Fatorar o resto da linha i (fazer $u_{ij} = -u_{ij}/u_{ii}$, para j = i+1, ..., tcol)

9. Copiar a coluna i da submatriz restante de B para L, considerando as permutações já efetuadas

10. Aplicar, nesta coluna i, os fatores das linhas anteriores ($l_{ji} = l_{ji} + l_{jk}.u_{ki}$, para j>i e k<i)

3. Para toda coluna j que restou do bloco h se este não for quadrado

1. Guardar em *icn* o índice da coluna da base que está na posição j (dado por *cpos[j]*)

2. Guardar em *luest* os elementos desta coluna relativos ao estoque

2. Para cada coluna j da base que só possui elementos não nulos no acoplamento

1. Guardar em *icn* a posição da coluna na base (*cpos[j]*)

2. Guardar em *luest* os seus elementos

3. Copiar os dados armazenados em *icn* para o fim do vetor *cpos*

4. Para todo bloco h entre 1 e *nbloc*

1. Para toda coluna j que sobrou à direita do bloco se este não for quadrado

1. Aplicar nos elementos das linhas do acoplamento desta coluna (guardados no vetor *luest*) os fatores das linhas superiores (já calculados)

5. Para toda linha de acoplamento i entre 1 e *nlest*-1

1. Escolher o pivô

2. Se o pivô for nulo

1. Então parar o programa (a base é singular)

3. Permutar os elementos das colunas *tlin+i* e *tlin+pivo*

4. Permutar *cpos[tlin+i]* e *cpos[tlin+pivo]*

5. Permutar *icn[i]* e *icn[pivo]*

6. Fatorar a submatriz restante de *B*

6. Para toda componente *i* de *icn*, *i* entre 1 e *nlest*

1. Encontrar *k* tal que *icn[k]* = *i*

2. Fazer *icn[i]* = *k*

SEÇÃO 5 A ATUALIZAÇÃO DA BASE

Uma vez dispondo da base decomposta em uma iteração anterior do simplex, podemos atualizá-la (evitando uma re-fatoração) seguindo o algoritmo abaixo.

1. Incrementar o número de atualizações já realizadas

2. Resolver $\omega^T.U = e^T_{sai}$

3. Descobrir a última posição não nula de ω, que chamamos *cult*

4. Criar uma lista ligada com os elementos da coluna *cult* de L

5. Para toda coluna *i* em que $\omega_i \neq 0$, partindo de *cult-1* até 1

1. Se $\omega_{cult} < \omega_i$

1. Então vamos zerar ω_{cult}:
    1. Criar uma nova matriz U^k cujo elemento não nulo acima da diagonal é $u^k_{i,cult} = -\omega_{cult}/\omega_i$
    2. Atualizar a coluna L_{cult} de L, fazendo $L_{cult} = L_{cult} - L_i.\omega_{cult}/\omega_i$
    3. Permutar as colunas L_i e L_{cult}
    4. Armazenar a permutação (*upos[k].p = i*)
2. Senão vamos zerar ω_i:
    1. Calcular $u^k_{i,cult} = -\omega_i/\omega_{cult}$
    2. Atualizar a coluna L_i de L, fazendo $L_i = L_i - L_{cult}.\omega_i/\omega_{cult}$
    3. Fazer *upos[k].p=cult* (não houve permutação)
3. Armazenar o valor de $u^k_{i,cult}$ em *u[ufimf+k]*
4. Armazenar sua linha (*upos[k].x = i*)
5. Armazenar sua coluna (*upos[k].y = cult*)

6. Somar a L a coluna que entra na base multiplicada por $\omega_{cult}.e^T_{cult}$
7. Subtrair de L a coluna que sai da base multiplicada pelo mesmo termo
8. Substituir o índice da coluna que sai pelo da que entra na base (*cabas[sai] = ent*)
9. Para toda linha i em que $l_{i,cult} \neq 0$, i entre 1 e *cult*-1
    1. Se $l_{i,i} < \mu.l_{i,cult}$ (μ é uma tolerância, no caso μ=10)
        1. Então vamos zerar $l_{i,cult}$:
            1. Criar uma nova matriz U^k cujo elemento não nulo supra-diagonal é $u^k_{i,cult} = -l_{i,cult}/l_{i,i}$
            2. Atualizar a coluna L_{cult} de L, fazendo $L_{cult} = L_{cult} - L_i.l_{i,cult}/l_{i,i}$

3. Fazer *upos[k].p=-cult* (não houve permutação)

2. Senão vamos zerar $l_{i,i}$:

1. Calcular $u^k_{i,cult} = -l_{i,i} / l_{i,cult}$

2. Atualizar a coluna L_i de L, fazendo $L_i = L_i - L_{cult} . l_{i,i} / l_{i,cult}$

3. Permutar as colunas L_i e L_{cult}

4. Armazenar a permutação (*upos[k].p* = *-i*)

2. Armazenar o valor de $u^k_{i,cult}$ em *u[ufimf+k]*

3. Armazenar sua linha (*upos[k].x* = *i*)

4. Armazenar sua coluna (*upos[k].y* = *cult*)

O passo 2 desta rotina requisita o procedimento **UtXY**, que será descrito na próxima seção.

SEÇÃO 6 RESOLUÇÃO DE SISTEMAS

Dadas as matrizes L e U atualizadas, vejamos como resolver

Sistemas do tipo B.z = y

Neste caso, o *pseudo-code* será

1. Resolver L.x = y
2. Resolver U.w = x
3. Fazer z = P.w ($z_{cpos[i]} = x_i$, para i = 1, ..., *tcol*)

Onde o passo 1 é executado pelo procedimento LXY e o passo 2 por UZX, fornecidos abaixo.

Procedimento LXY

1. Fazer x = y
2. Para toda componente j da base, começando por $j = 1$
   1. Fazer $x_j = x_j / l_{j,j}$
   2. Para toda componente i da base, $i > j$
      1. Fazer $x_i = x_i - x_j / l_{i,j}$

Procedimento UZX

1. Para todas as matrizes U^k acrescentadas nas atualizações, partindo da última
   1. Seja *ut* = *upos[k]*
   2. Se *ut.p* < 0
      1. Então somar a $x_{ut.x}$ o termo $x_{ut.y}.u^k_{ut.x,ut.y}$
      2. Senão somar a $x_{ut.y}$ o termo $x_{ut.x}.u^k_{ut.x,ut.y}$
   3. Se *ut.p* ≠ *ut.y*
      1. Então permutar $x_{ut.p}$ e $x_{ut.y}$
2. Para toda componente i da base, começando pela última
   1. Para toda componente j da base, $j > i$
      1. Fazer $x_i = x_i + x_j.u_{ij}$

Sistemas do tipo $B^T.z = y$

Para sistemas que envolvem a transposta da base teremos que:

1. Fazer $x = P^T y$ ($x_i = y_{cpos[i]}$, para i = 1, ..., *tcol*)
2. Resolver $U^T.w = x$
3. Resolver $L^T.z = w$

Aqui, as subrotinas que executam os passos 2 e 3 são, respectivamente, **UtXY** e **LtZX**.

Procedimento UtXY

1. Para toda componente j da base, começando por $j = 1$
   1. Para toda componente i da base, $i > j$
      1. Fazer $x_i = x_i + x_j.u_{ij}$
2. Para todas as matrizes U^k acrescentadas nas atualizações, partindo de $k = 1$
   1. Seja *ut* = *upos[k]*
   2. Se *ut.p* ≠ *ut.y*
      1. Então permutar $x_{ut.p}$ e $x_{ut.y}$
   3. Se *ut.p* < 0
      1. Então somar a $x_{ut.y}$ o termo $x_{ut.x}.u^k_{ut.x,ut.y}$
      2. Senão somar a $x_{ut.x}$ o termo $x_{ut.y}.u^k_{ut.x,ut.y}$

Procedimento LtZX

1. Para toda componente j da base, começando pela última
    1. Para toda componente i da base, $i > j$
        1. Fazer $x_j = x_j - x_i / l_{i,j}$
    2. Fazer $x_j = x_j / l_{j,j}$
2. Fazer $z = x$

SEÇÃO 7 ESCALAMENTO

Sabemos que os erros de arredondamento que se acumulam sobre os valores dos elementos das matrizes triangulares durante a decomposição ou atualização da base são proporcionais à magnitude destes mesmos elementos. Assim, até a melhor estratégia de pivoteamento poderá falhar se os elementos da base tiverem valores (em módulo) muito discrepantes.

O artifício comumente empregado para contornar isto consiste em escalar a matriz tecnológica do problema antes de resolvê-lo, ou seja, dividir seus elementos por certos valores

de forma a permitir que haja uma maior uniformidade entre else quando participarem das decomposições.

O método de escalamento mais popular é denominado *de equilíbrio* e consiste apenas em dividir cada linha (ou coluna) da matriz pelo seu elemento de maior valor absoluto.

E, a despeito de existirem outros métodos comprovadamente melhores que este, preferimos adotá-lo aqui em virtude de ter sido ele empregado juntamente aos outros algoritmos com os quais o que expusemos será comparado no capítulo que se segue.

Assim, iremos apenas escalar a matriz tecnológica por linhas, segundo o *pseudo-code* abaixo:

1. Para cada linha *i* da matriz A
    1. Encontrar o elemento de maior valor absoluto, *escli*
    2. Dividir todos os elementos da linha por *escli*
    3. Dividir o termo correspondente no vetor *b* por *escli*
    4. Para cada variável de folga das restrições canalizadas
        1. Se esta variável corresponder à linha *i*
            1. Dividir os valores de seus limites inferior e superior por *escli*
2. Encontrar o elemento de maior valor absoluto da função objetivo, denominado *ctobj*
3. Dividir os elementos da função objetivo por *ctobj*

O uso contínuo de um processo de atualização da base produz efeitos nada desprezíveis sobre o comportamento e o desempenho geral do simplex.

Alguns destes efeitos são inerentes ao algoritmo utilizado, mas existem também aqueles que são fruto de decisões tomadas quando do desenvolvimento do programa de computador.

Apenas para exemplificar, citemos a grande variedade de estruturas de dados passíveis de serem adotadas, induzindo, em cada caso, um maior ou menor consumo de memória e de tempo.

Tantos outros pontos dúbios foram tratados com superficialidade e brevidade ao longo do texto com o objetivo de evitar que se entendesse o emprego de uma ou outra alternativa como algo intrínseco ao algoritmo que a contém.

No entanto, o momento parece, agora, oportuno para retornarmos a alguns desses pontos, justificando, em cada caso, a adoção de uma opção determinada.

Comecemos analisando os limitantes que podem ser adotados nos pivoteamentos feitos na atualização, descritos ao final da seção II.4.

Determinação do primeiro limitante da atualização

No item relativo às **Estratégias de pivoteamento** da seção II.4, dois eram os limitantes utilizados durante a transformação da base atualizada em novas matrizes triangulares.

O primeiro deles era empregado quando querímos eliminar os elementos não nulos da primeira linha da matriz

$$\begin{bmatrix} \omega^T \\ L \end{bmatrix}$$

Dadas as colunas k e m desta matriz, com m situada a direita de k (veja a figura II.4.4), existiam, então, duas alternativas para tornar nulo ω_k^T:

1 Fazer $\tilde{L}_k = L_k - \frac{\omega_k}{\omega_m} L_m$, ou

2 Fazer $\tilde{L}_m = L_m - \frac{\omega_m}{\omega_k} L_k$ e permutar as colunas m e k

A decisão por uma delas era tomada em função do termo

$$\mu = \omega_k / \omega_m \qquad \text{(III.7.1)}$$

ao qual associamos um limitante $\bar{\mu}$, de tal forma que a primeira opção era adotada se e somente se $\mu < \bar{\mu}$.

Analisemos, agora, quais poderiam ser as conseqüências do emprego de diversos valores deste limitante.

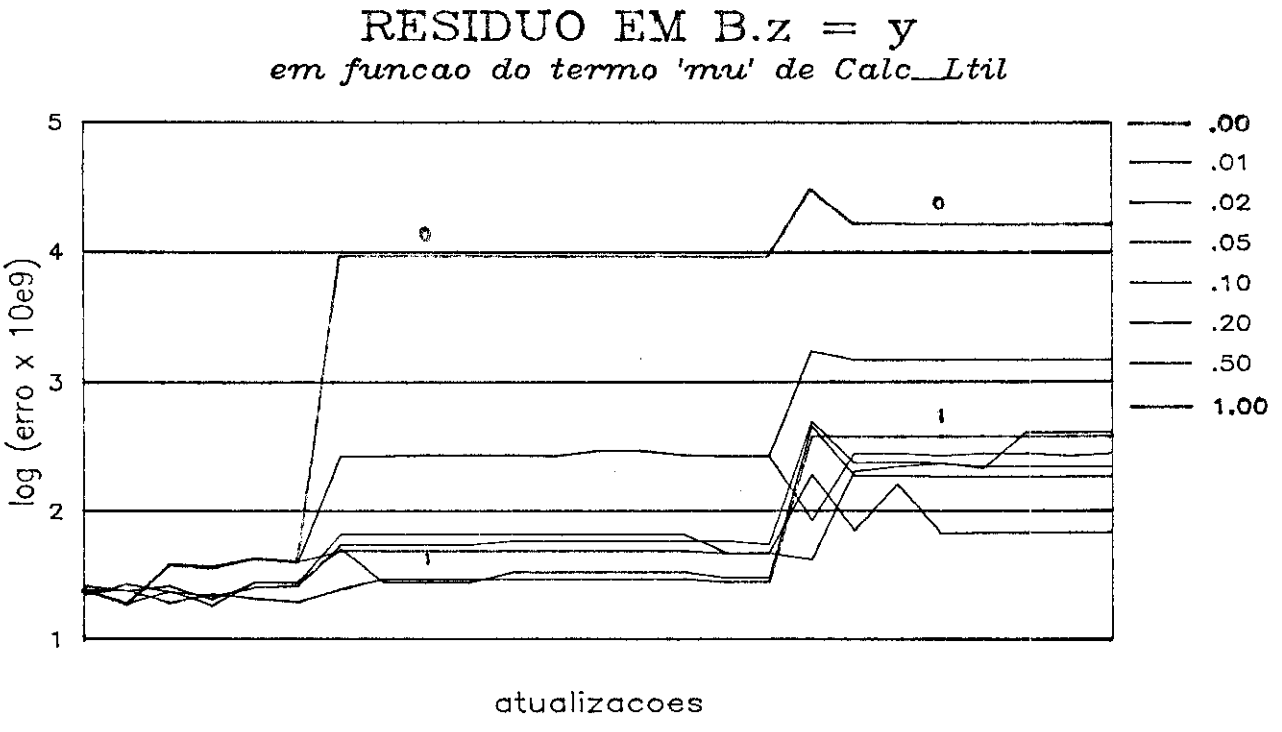
Os resíduos na solução de sistemas

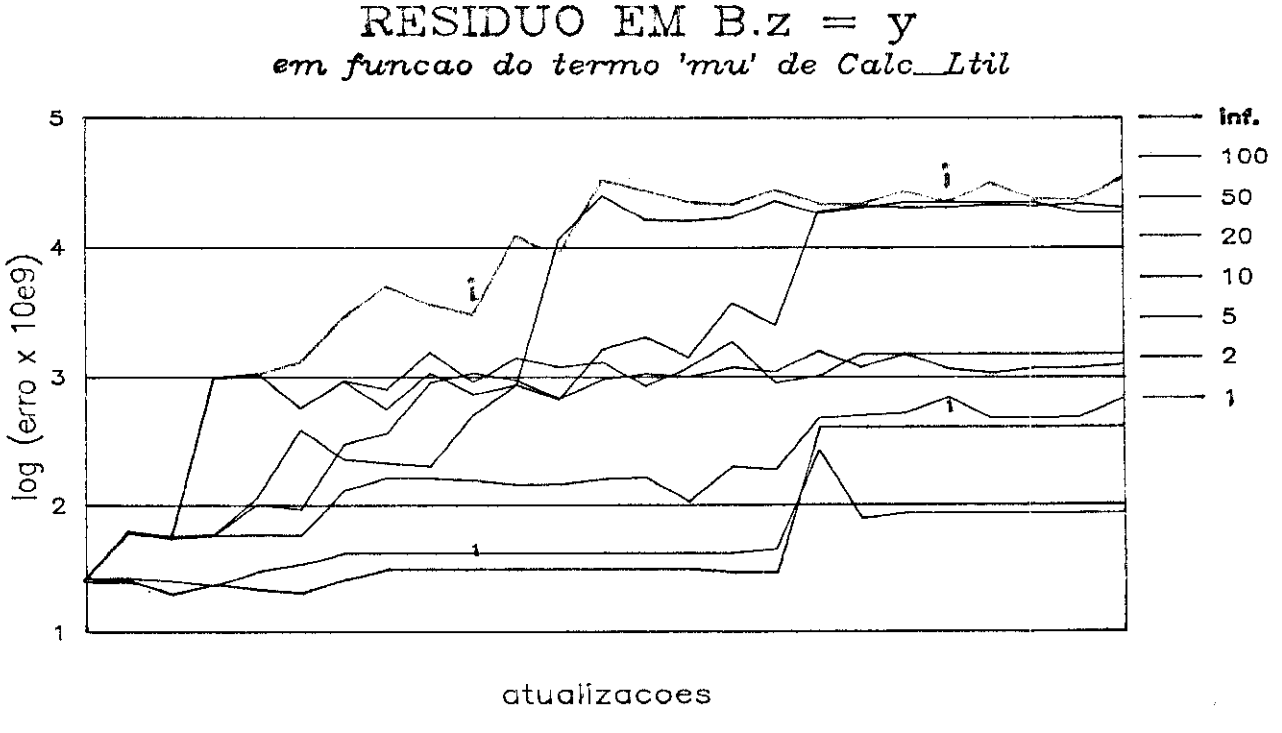
Atribuidos alguns valores arbitrários para $\bar{\mu}$, foram feitas experiências aleatórias nas quais diversas atualizações eram executadas sucessivamente, de modo a permitir que se evidenciasse, em cada caso, o efeito progressivo destas sobre os erros de arredondamento encontrados na resolução de sistemas. Os resultados estão expostos nos gráficos III.7.1 e III.7.2.

E se reza a teoria que os melhores resultados são obtidos para valores de μ próximos de 1, este fato foi, de uma forma geral, observado, embora o menor resíduo encontrado não tenha correspondido à curva de $\mu = 1$ durante todo o tempo. Mas parece claro que, como comportamento médio, os erros tendem a aumentar quando nos aproximamos dos valores extremos do limitante[4].

---

[4]Observa-se que nos casos em que $\mu < 1$ a segunda opçaõ de pivoteamento é favorecida, situação que se inverte quando μ é maior que a unidade.





- Gráficos III.7.1 e III.7.2 -

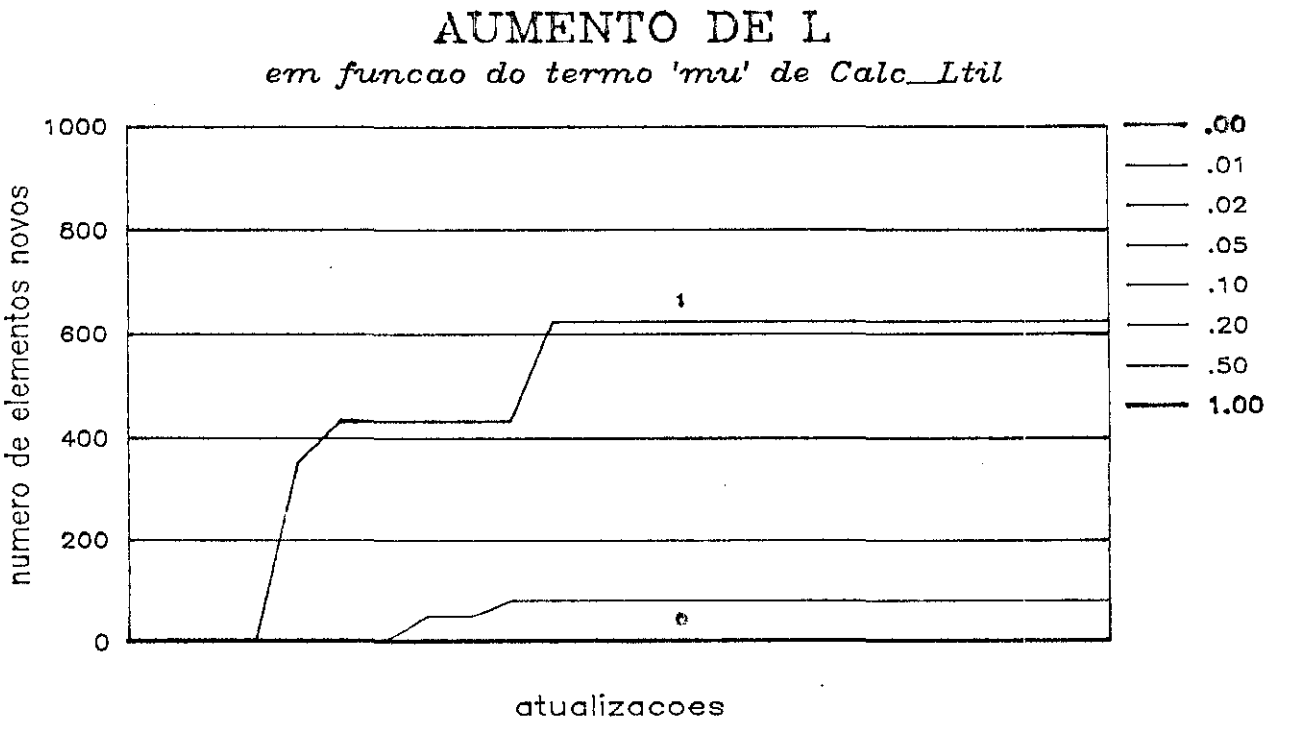
O aumento da matriz L

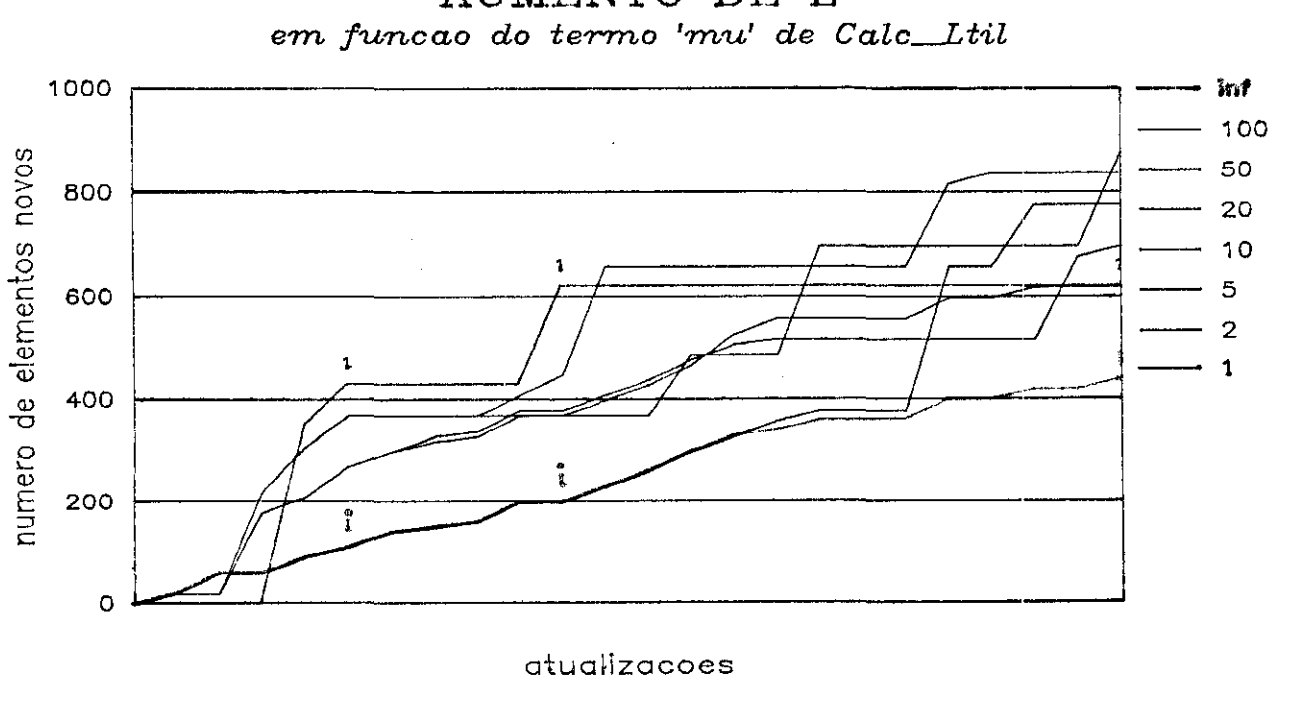
No que diz respeito ao aumento específico do número de elementos não nulos de L, podemos esperar que, à medida em que as atualizações se sucedem, este seja menor para valores muito pequenos de μ, embora freqüentemente isso não seja observado.

Mas as nossas experiências mostram, surpreendentemente, um comportamento muito melhor que o previsto para μ entre 0 e 0,5 e para $\mu = \infty$ (gráficos III.7.3 e III.7.4).

Nos casos em que $\mu < 0.5$ nenhum elemento não nulo foi criado em qualquer atualização, quando se esperava um aumento mesmo que discreto da matriz. Da mesma forma, o crescimento de L para $\mu = \infty$ foi menor que o obtido para valores menores do limitador.

Outra surpresa foi o fato do pior resultado ter sido obtido para $\mu = 1$, o que, mesmo não sendo os dados suficientes para que se possa tirar conclusões definitivas, certamente contribui para confirmar a hipótese, aventada no capítulo II, de que não é possível assegurar que o consumo de memória pode ser reduzido em função da alteração deste limitador.





- Gráficos III.7.3 e III.7.4 -

O crescimento da matriz U

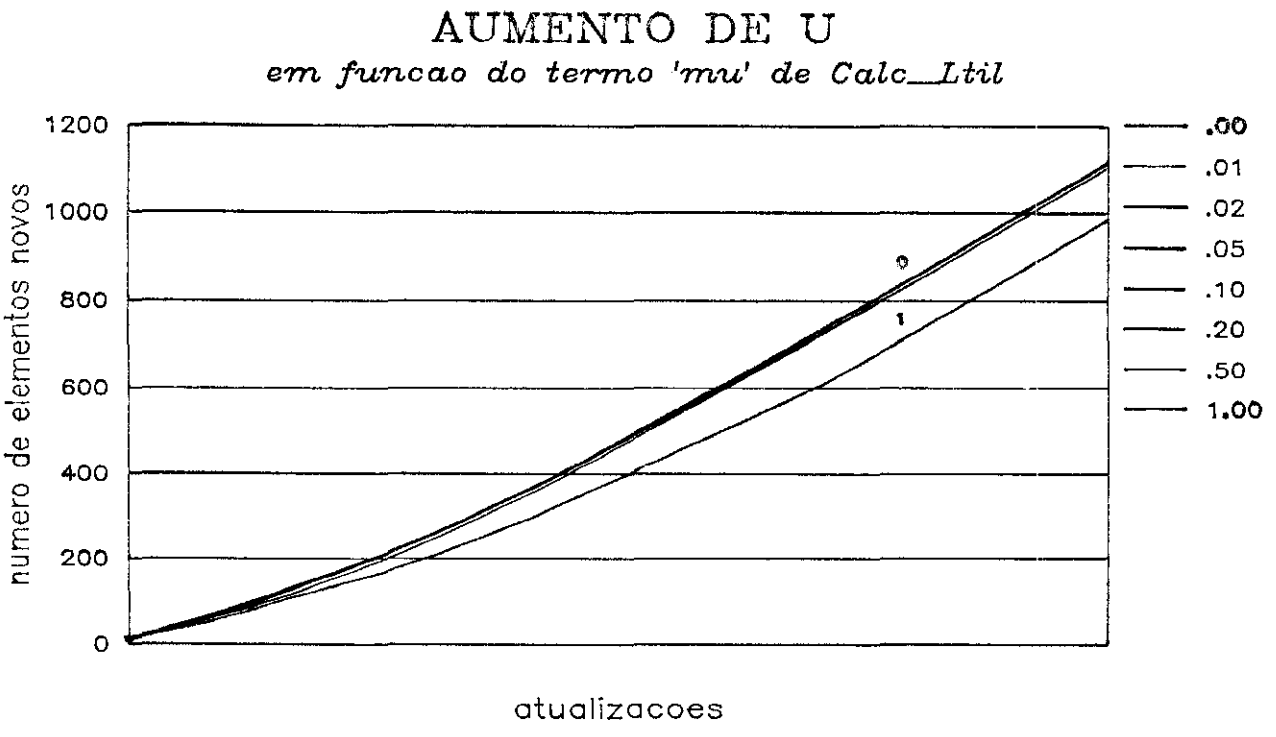
Também foi analisado o efeito da variação de $\bar{\mu}$ sobre o número de elementos de U, estando os resultados expostos nos gráficos II.7.5 e III.7.6.

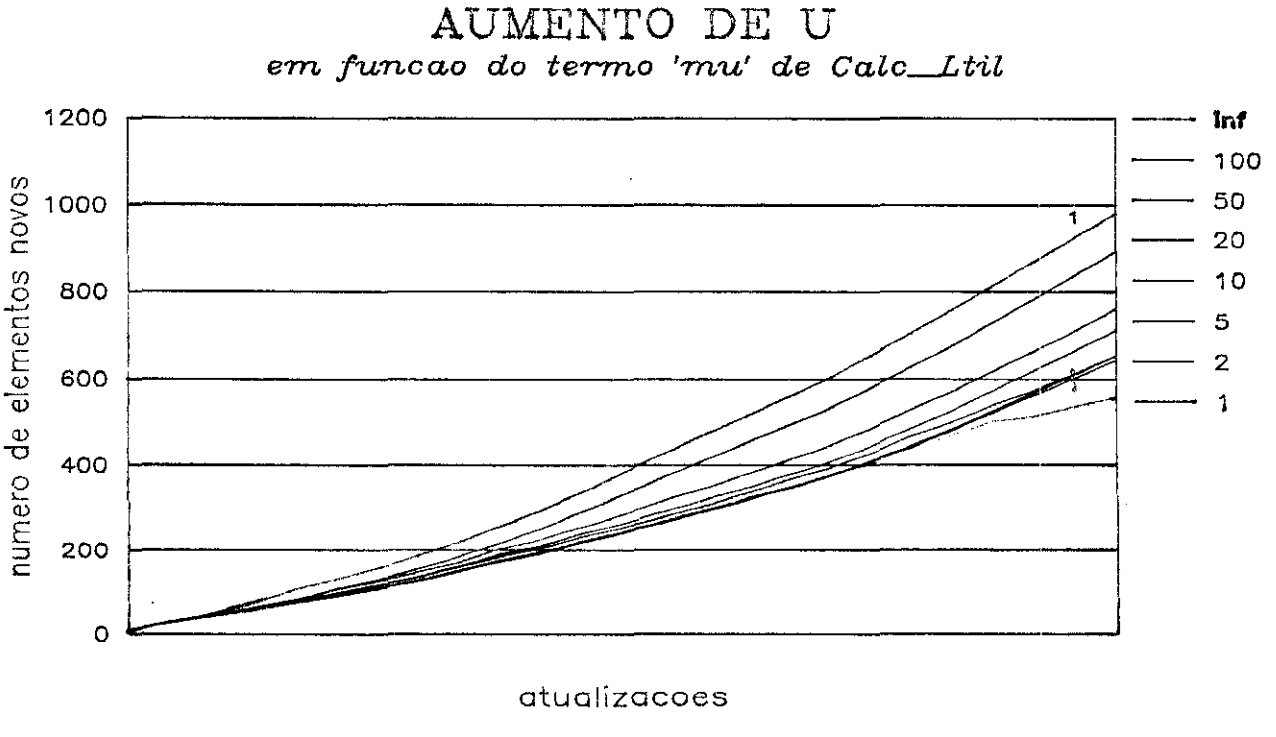
Neste caso, verificou-se que valores pequenos de μ costumam induzir um crescimento maior da matriz U, o que se explica facilmente pelo fato de que, quando se adota com mais freqüência a segunda opção de pivoteamento, torna-se mais densa a coluna de $\tilde{L}$ que, ao final da eliminação dos elementos de ω, possui elementos não nulos acima da diagonal.

É interessante observar também o comportamento quase linear do aumento de U, bem diferente dos saltos existentes nos gráficos referentes à matriz L.

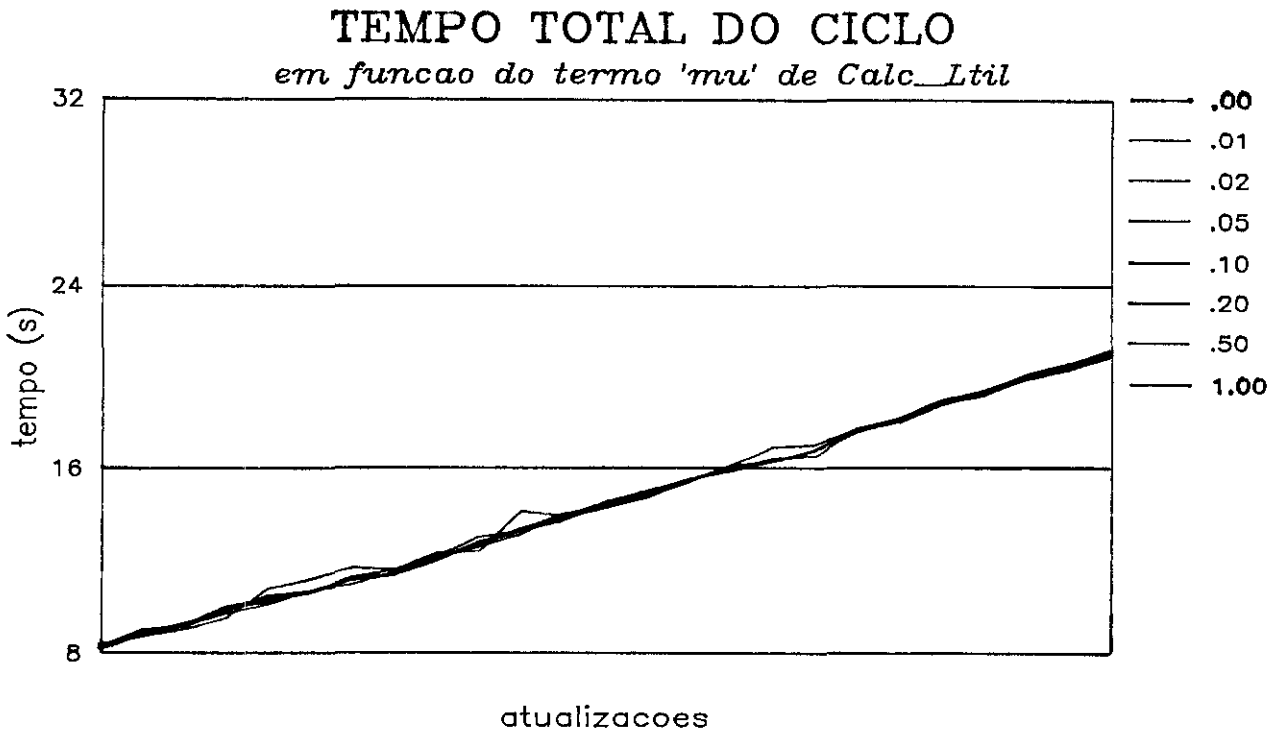
O tempo consumido em cada ciclo do simplex

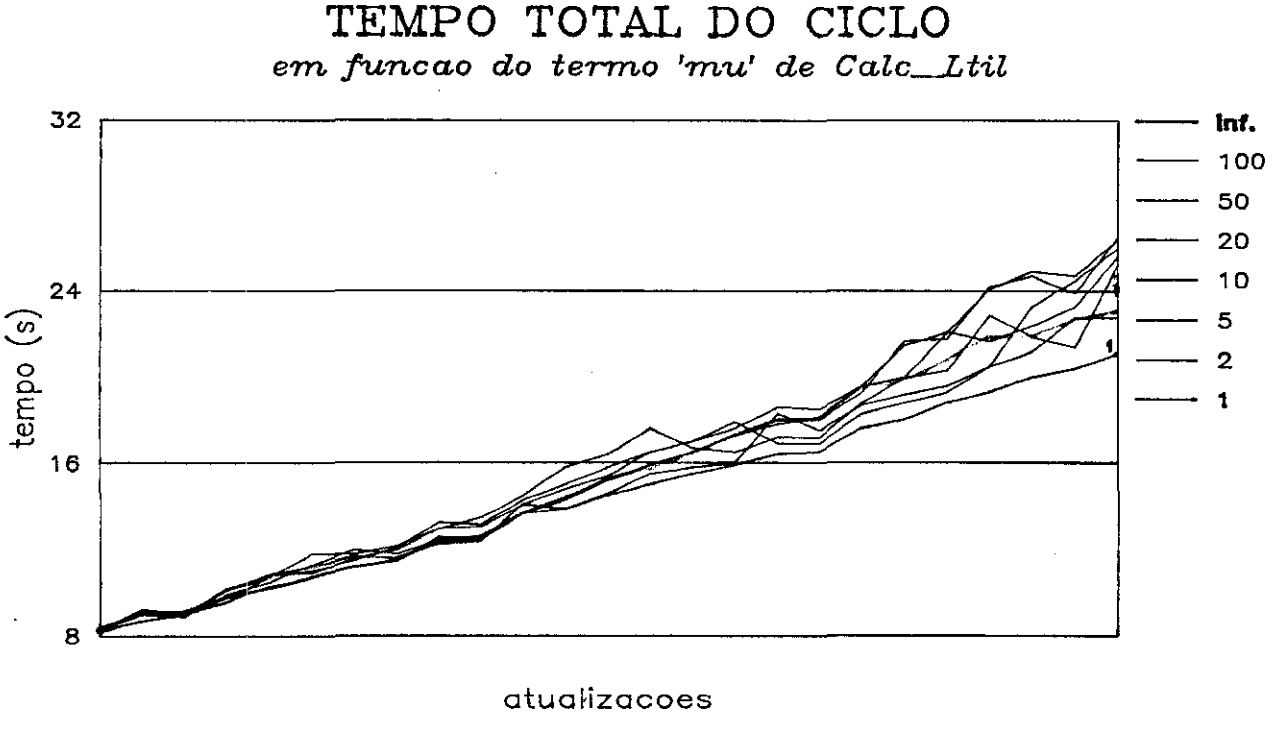
Com o uso da estrutura de dados proposta, que combina listas ligadas com memória estática, o tempo de cada ciclo do simplex tornou-se menos sujeito a variações em função do limitante $\bar{\mu}$, como se pode deduzir dos gráficos III.7.7 e III.7.8.





- Gráficos III.7.5 e III.7.6 -





- Gráficos III.7.7 e III.7.8 -

E se poderíamos esperar que, a medida que aumentássemos o número de elementos de L, maior fosse o tempo gasto na atualização, este efeito foi bastante reduzido dada a facilidade com que se pode permutar colunas descritas por listas ligadas.

Por sua vez, a resolução de sistemas, que também é afetada pelo crescimento de L e U, por consumir pouco tempo, pouco contribuiu para uma alteração global significativa no desempenho do programa.

O valor adotado para o limitante

Para preservar a estabilidade numérica do problema e sem ter como determinar antecipadamente o impacto do uso de um determinado valor para este limitante sobre o consumo de memória, optou-se por empregar o valor 1.

Deseja-se, com isso, contribuir para que o número de atualizações sucessivas permitido pelo critério de estabilidade, medido através do procedimento CalcNorma, seja maior, compensando um eventual aumento indesejado das matrizes.

Além disso, o tempo de execução do programa sofreria mais com a necessidade de fatorações sucessivas em virtude de problemas de estabilidade do que em função de matrizes ligeiramente maiores.

Cabendo salientar, ainda, que não se pode afirmar, em função do observado, que utilizando o valor 1 teremos um maior consumo de memória sempre.

Determinação do segundo limitante da atualização

O último passo da atualização também consiste em eliminar elementos de uma matriz — $L^{\#}$ — que, por sua vez, possui elementos acima da diagonal em apenas uma coluna (digamos, a de índice k).

A diferença está que, aqui, o uso de limitantes diferentes pode causar efeitos bem diversos.

Valores de $\bar{\mu}$ menores que 1, por exemplo, além de não melhorarem a estabilidade, costumam provocar um consumo muito grande de memória.

Vejamos, então, como se comporta o algoritmo quando variamos $\bar{\mu}$ entre 1 e infinito.

O aumento do consumo de memória

Para este limitante, como foi dito no capítulo II, o emprego de valores baixos está intrisecamente ligado a um aumento exagerado da matriz L.

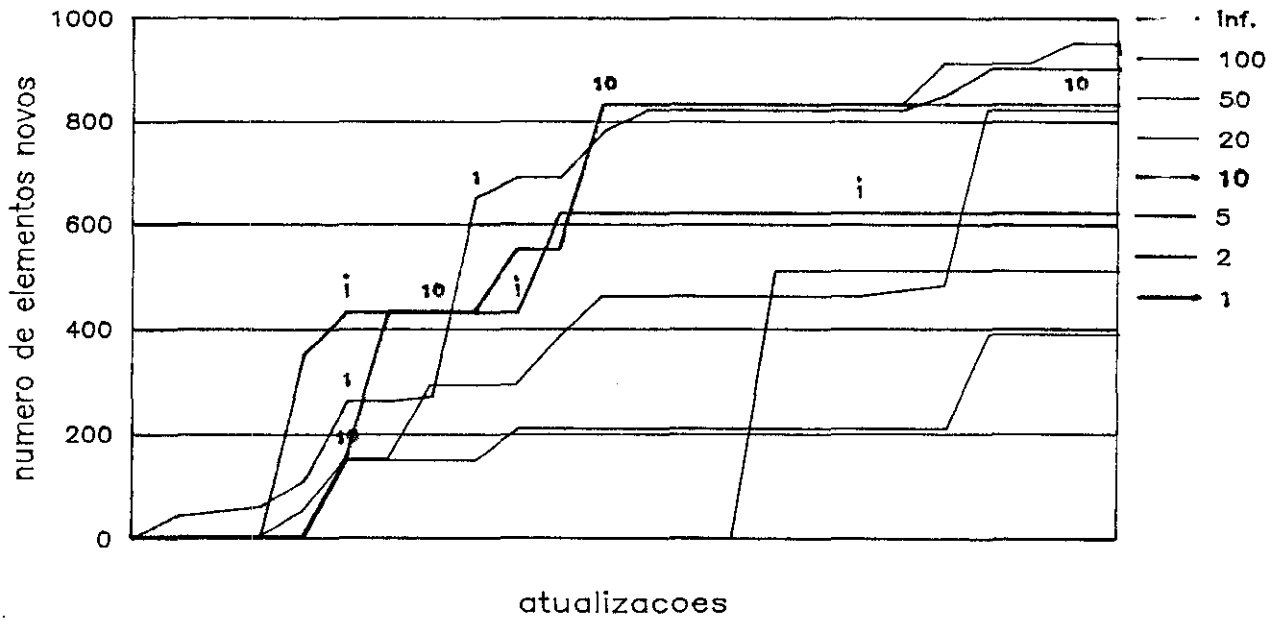
Esta afirmação não fica tão clara quando observamos os resultados experimentais (figuras III.7.9 e III.7.10). Eles apontam, é verdade, nesta direção, mas não com tanta evidência.

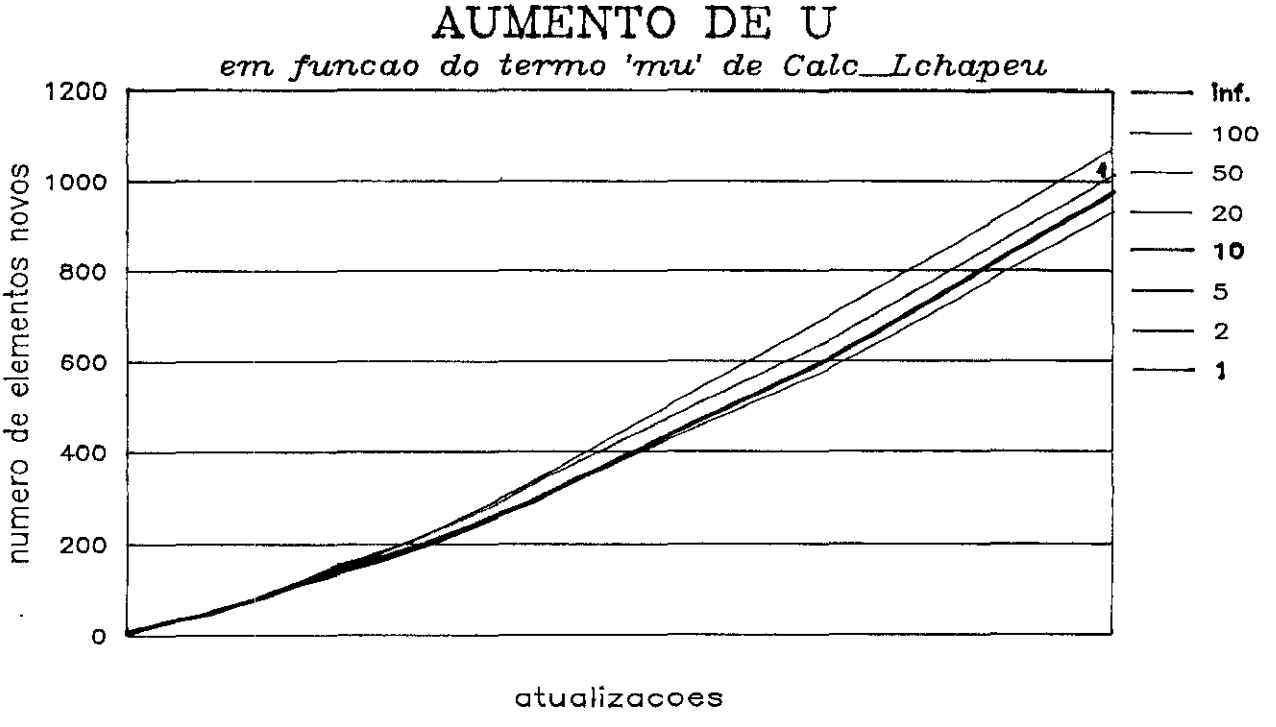
Atribuimos isto ao fato de que os resultados muito ruins só são obtidos em situações extremas, o que não costuma ocorrer com tanta freqüência.

Quanto ao crescimento de U, este ainda se dá através de curvas bastante suaves e sem que se observe grande dispersão entre elas. Em cada atualização, as diferenças no consumo de memória são devidas principalmente a alguns elementos da matriz L criados nas iterações anteriores.

O tempo consumido no ciclo do simplex

As diferenças encontradas no tempo consumido por cada ciclo do simplex somente podem ser atribuidas ao número e à posição dos elementos criados nas matrizes triangulares, principalmente em L.





- Gráficos III.7.9 e III.7.10 -

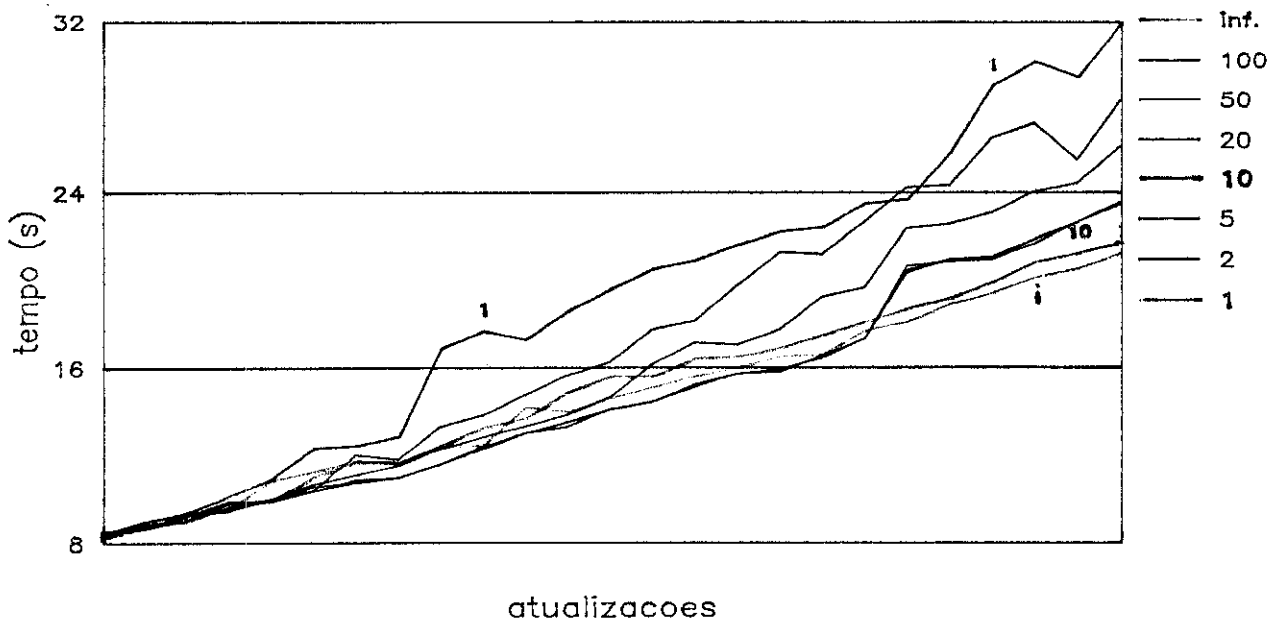
Assim, para os casos em que pouco cresceu esta matriz — geralmente para valores grandes de $\bar{\mu}$ — menor será o tempo do ciclo. Esta conjetura é, de uma maneira geral, confirmada pelo gráfico III.7.11.

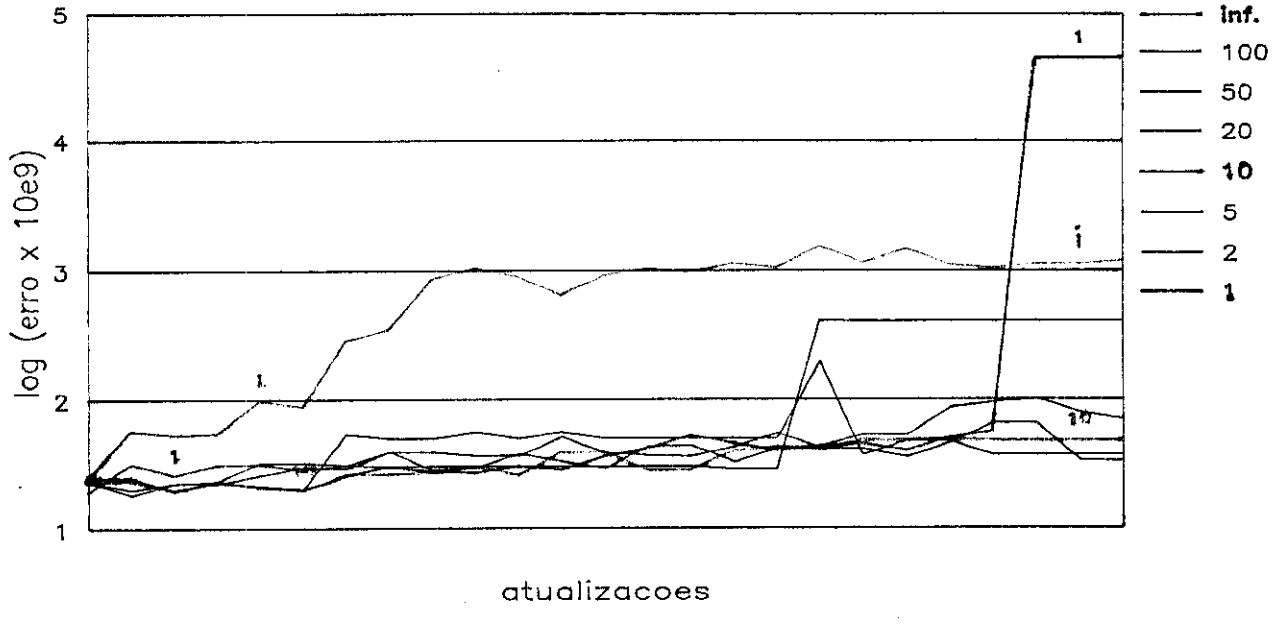
Os resíduos na resolução de sistemas

Quanto ao erros de arredondamento provocados na resolução de sistemas, também aqui a teoria sugere que para reduzí-los devemos apelar para o tradicional pivoteamento parcial, ou seja, fazer $\mu = 1$.

O gráfico III.7.12, mais uma vez, mostra que isto ocorre durante a maior parte do tempo. Apenas nas últimas atualizações observou-se um aumento expressivo e inesperado do resíduo para $\mu = 1$, o que não pudemos atribuir a nenhuma causa evidente e que iremos tratar como fenômeno atípico.

Mas, convém lembrar que os erros de arredondamento também aumentam proporcionalmente ao tamanho das matrizes, donde não seria exagero esperar que, para valores mais baixos de μ, houvesse um crescimento deles, a despeito do critério de estabilidade nos pivoteamentos.





- Gráficos III.7.11 e III.7.12 -

O valor adotado para o limitante

Para mater um equilíbrio entre consumo de memória e estabilidade numérica, resolvemos adotar o valor 10 para o limitante, valor este que também tem sido recomendado em casos semelhantes por diversos especialista no assunto.

Com isso, tentamos evitar as situações extremas em que as matrizes crescem enormemente, sem contudo permitir erros de arredondamento capazes de comprometer nosso algoritmo.

E é interessante conferir, apenas como curiosidade, nos últimos gráficos, os resultados referentes a este valor.

Determinação da freqüência das fatorações

Uma última preocupação que nos ocorreu ao escrever o programa diz respeito à decisão que, a cada ciclo do simplex, deve ser tomada entre atualizar a base e refatorá-la.

A primeira vista, parece-nos razoável evitar ao máximo a fatoração, mas haverá um momento em que ela pode ser conveniente ou mesmo necessária se considerarmos que

1 Pode não existir memória disponível para armazenar os elementos que serão criados durante a atualização;

2 Os resíduos encontrados na resolução dos sistemas podem ser excessivos; ou

3 Uma nova atualização irá provocar um consumo maior de tempo pelos próximos ciclos do simplex.

Dediquemos um pouco mais de atenção a cada um destes pontos.

Os resíduos encontrados na resolução de sistemas

Em nosso programa, os erros de arredondamento são verificados a cada 10 atualizações, o que significa que, ao fim deste intervalo, determinamos o maior resíduo absoluto encontrado no cálculo do vetor solução do problema e o comparamos com um limitante pré-estabelecido (que depende da precisão do computador), refatorando a base sempre que este for ultrapassado.

Este critério tem garantido bons resultados do ponto de vista numérico e é de uso corrente, donde nos absteremos de detalhá-lo.

O tempo gasto pelas atualizações

Por sua vez, o tempo consumido em cada ciclo do simplex merece um estudo mais aprofundado, pois afeta diretamente o desempenho do algoritmo.

Preferimos, então, realizar novas experiências, agora com problemas reais, para tornar possível determinar, com mais certeza, o melhor momento para uma nova fatoração.

Primeiramente, elaboramos o gráfico III.7.13, que ilustra a economia relativa obtida pelo uso de cada uma das atualizações sucessivas em função do tempo consumido por uma nova fatoração, para um problema cuja matriz tecnológica possuia 14 blocos e apenas 2 restrições de estoque.

Para tanto, a resolução do problema foi dividida nas fases tradicionais do simplex[5], cada uma delas sendo também dividida ao meio. Conseguiu-se, assim, observar que o algoritmo se comporta de maneira diferente em uma e outra fase.

Na fase 1, em que a maioria das colunas que deixa a base corresponde a variáveis artificiais, as atualizações costumam fornecer uma economia quase constante, que decresce muito lentamente em decorrência apenas do aumento da matriz U

---

[5]Na fase 1 procuramos uma solução viável e na fase 2 a solução ótima para o problema.

(que pouco afeta as atualizações subsequentes).

Já na fase 2 a economia relativa se reduz bastante a cada iteração, refletindo o crescimento da matriz L.

Os vales profundos do gráfico correspondem às iterações em que o vetor solução do problema é recalculado e que a norma do resíduo é verificada. Pode-se-á, nos gráficos seguintes, constatar o efeito destes procedimentos sobre o tempo de execução do ciclo.

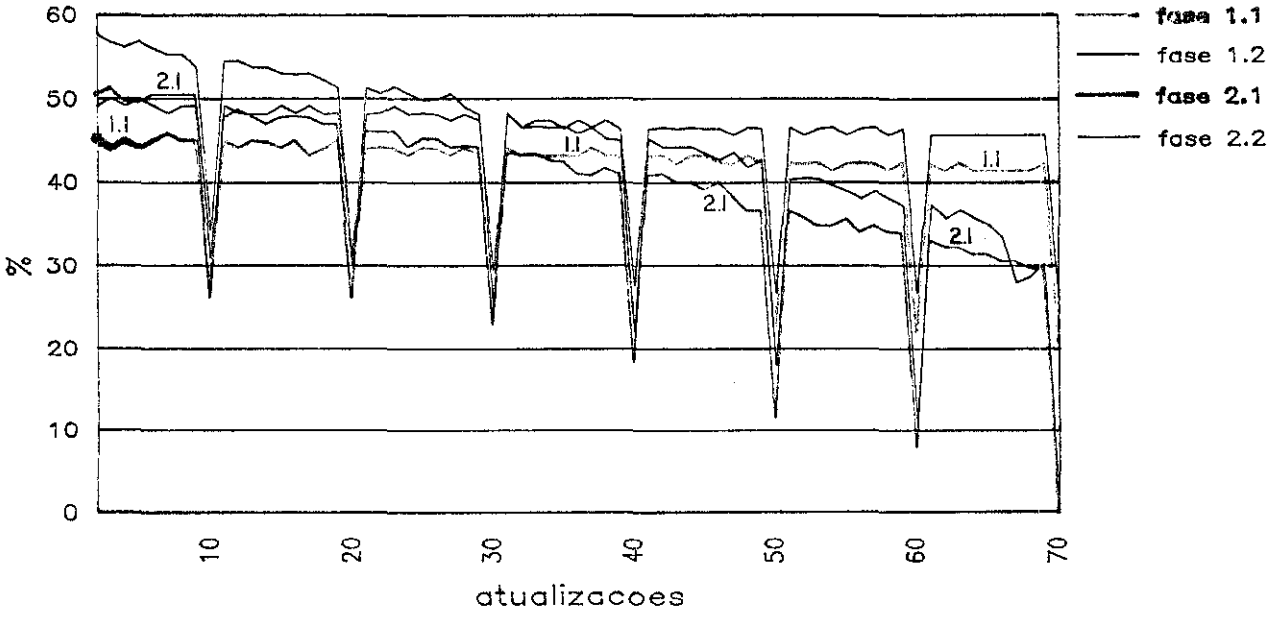
Passemos agora ao gráfico III.7.14, que mostra como se acumula, à medida que cresce o número de iterações, a economia global do programa obtida quando se evita refatorar continuamente a base.

Podemos ver que há um aumento rápido desta economia durante as primeiras atualizações, seguido de uma queda que está associada aos vales do gráfico III.7.13. A partir de então as curvas correspondentes à fase 1 permanecem quase horizontais, enquanto as da fase 2 sobem lentamente mais um pouco, até decairem também de forma quase retilínea.

O gráfico nos mostra, assim, que ao final das 9 primeiras iterações o emprego da rotina de atualização reduziu à metade, por exemplo, o tempo do programa na segunda parte da fase 2 do simplex, mas se continuarmos atualizando a base sem refatorá-la por mais 60 iterações esta economia diminuirá para pouco mais de 40 por cento.

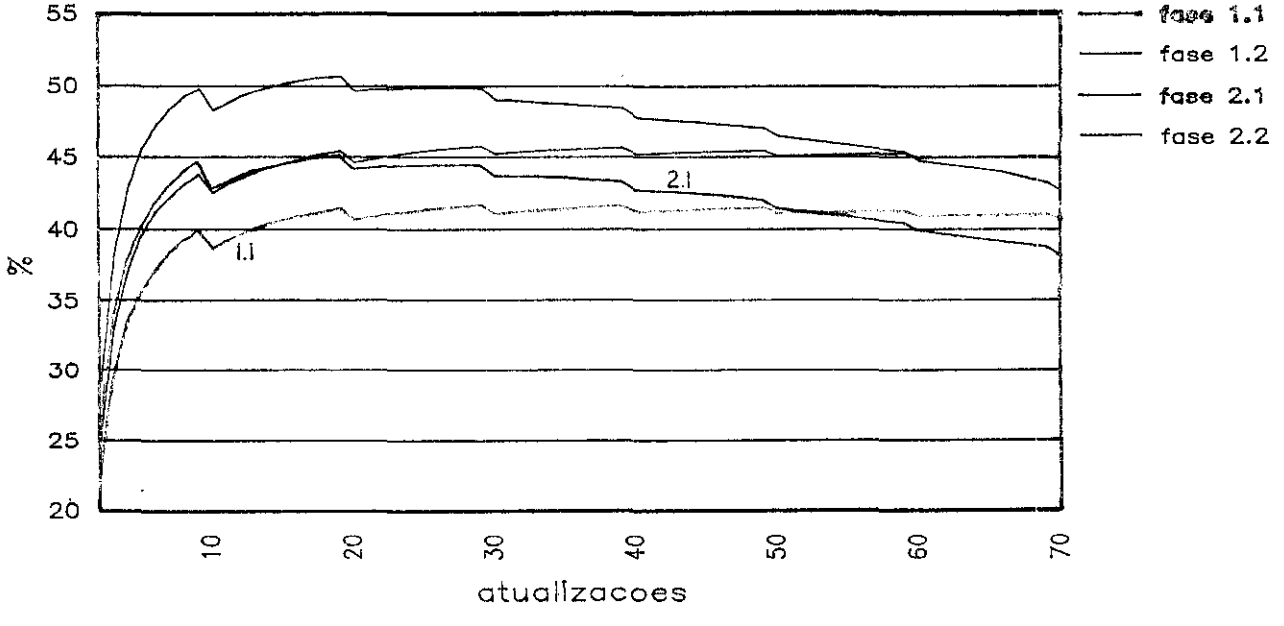
ECONOMIA DE CADA ATUALIZACAO

Problema com 14 blocos e 2 linhas de acoplamento



ECONOMIA DE TEMPO ACUMULADA

Problema com 14 blocos e 2 linhas de acoplamento



- Gráficos III.7.13 e III.7.14 -

Somos levados a crer, assim, que em algum momento entre as iterações 10 e 70 deveríamos ter refatorado a base.

Este momento determinamos através do seguinte *fator de economia de tempo*:

$$\phi_k = \frac{m \cdot \sum_{i=1}^{k} T_i}{k \cdot \sum_{i=1}^{m} T_i} \qquad \text{(III.7.2)}$$

onde k é o número de iterações do simplex efetuadas a partir da última fatoração, m é o índice da última iteração em que o fator foi menor que 1, e T_i é o tempo gasto na iteração i.

Para exemplificar como é obtido este fator, vamos supor que estamos na trigésima iteração desde a última fatoração (neste período a base foi atualizada 29 vezes), tendo o programa consumido desde então 325 segundos, e a última iteração na qual o valor de ϕ era menor que 1 foi a de número 19, ocorrida 210 segundos desde a fatoração. Assim, o novo fator irá valer: (19x325)/(30x210), ou 0.98016.

Calculado após cada atualização[6], o fator ϕ será menor que 1 quando tiver sido conveniente executá-la em lugar de uma

---

[6]Para a primeira atualização consideramos que $k = 2$ e $m = 1$.

refatoração. Por outro lado, valores muito maiores que a unidade indicam que esta última tornaria o algoritmo mais eficiente.

Fica claro, assim, que este fator não determina antecipadamente qual alternativa deve ser adotada, mas apenas indica *a posteriori* se opção escolhida era ou não a mais econômica.

Os gráficos III.7.15, III.7.16 e III.7.17 mostram, para alguns exemplos de dimensões variadas, os valores de ϕ a medida em que se sucedem as atualizações. Assim, por exemplo, para o problema com 14 blocos e duas linhas de estoque, observamos que na primeira parte da fase 1 poderíamos ter executado 59 atualizações sucessivas antes de uma nova refatoração[7]. Já na fase 2 não teria sido recomendável atualizar a base por mais de 29 iterações sucessivas (um comportamento coerente com os gráficos III.7.13 e III.7.14).

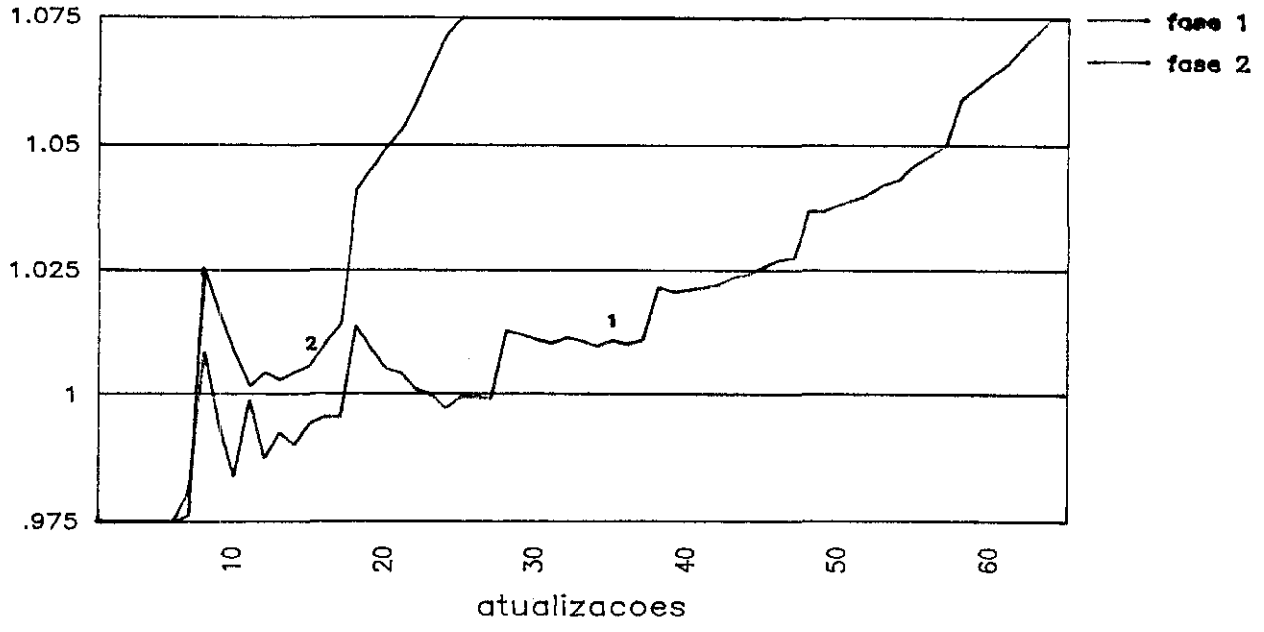
E apesar deste fator se referir apenas às iterações já ocorridas, temos a pretensão de empregá-lo para determinar como devemos agir na próxima iteração, como veremos a seguir.

---

[7]É interessante reparar que neste caso é menos econômico atualizar a base por 58 iterações sucessivas do que fazê-lo por apenas 49 iterações. Mas ainda assim a quinquagésima atualização se justifica, pois ϕ_{59} é menor que 1.

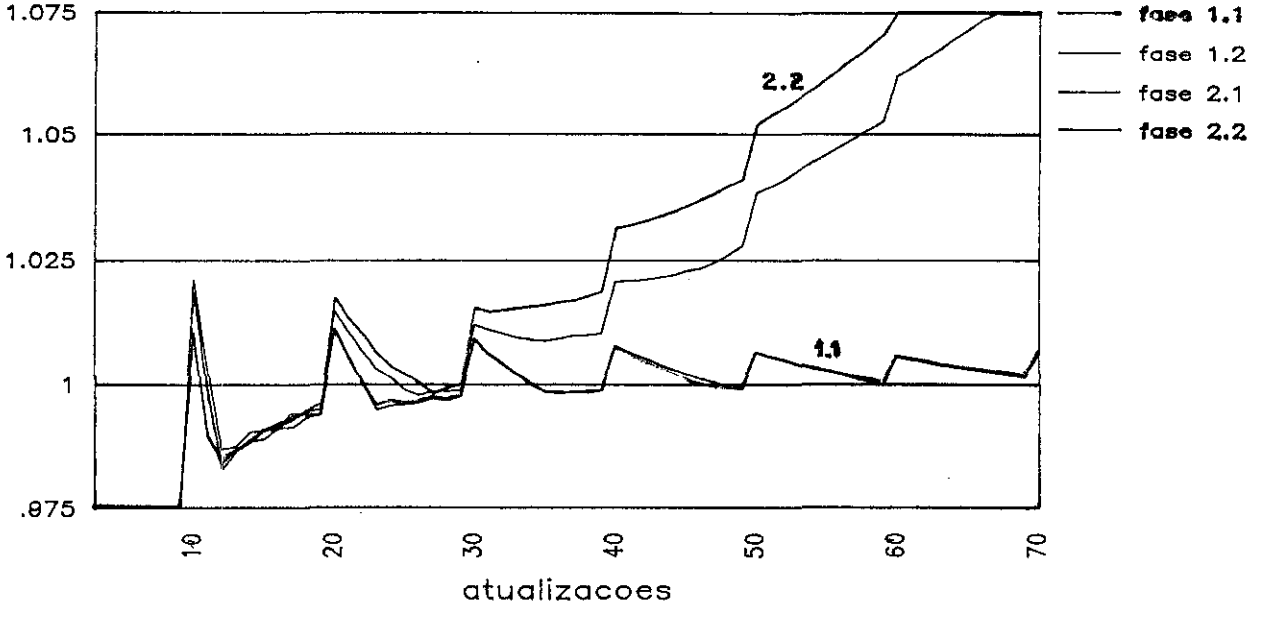
FATOR DE ECONOMIA DE TEMPO

Problema com 4 blocos e 2 linhas de acoplamento

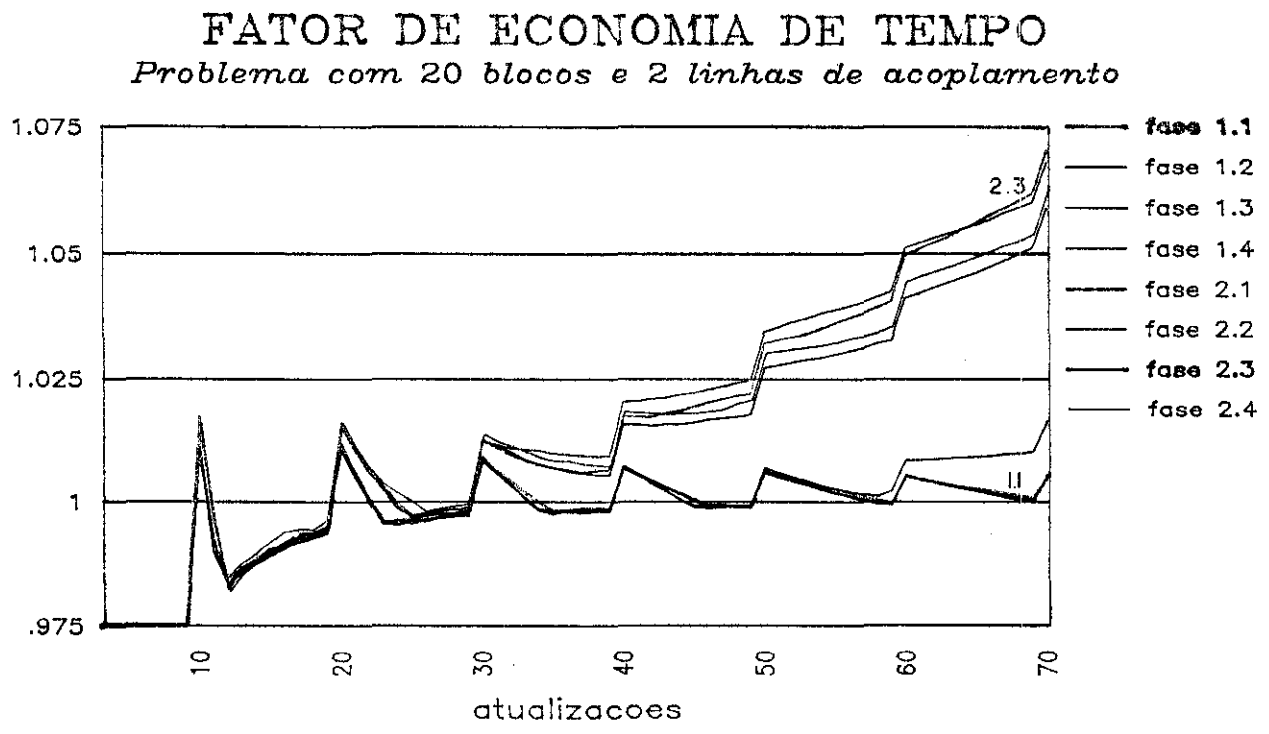


FATOR DE ECONOMIA DE TEMPO

Problema com 14 blocos e 2 linhas de acoplamento



- Gráficos III.7.15 e III.7.16 -



- Gráfico III.7.17 -

Podemos aproveitar os gráficos III.7.15, III.7.16 e III.7.17 acima para tirar duas conclusões importantes:

I Em primeiro lugar, consideremos que a norma do resíduo encontrado no cálculo do vetor solução do problema seja verificada ciclicamente a cada υ iterações depois de uma fatoração (ou seja, nas iterações υ, 2υ, 3υ, etc). Nota-se que, nestes casos, o valor de ϕ é *muito alto* (*correspondendo aos* picos dos gráficos), mas este fato não deve ser levado em conta pois nas iterações subsequentes ϕ tende a diminuir;

2 Normalmente quando $\phi_{k-1} < 1$, este comportamento tende a se repetir na iteração k, exceto no caso em que $k = p.v - 1$, $p = 1, 2, 3, \ldots$ (ou seja, se $k = v-1, 2v-1$, etc).

Com base nessas observações, foi possível estabelecer, finalmente, o seguinte critério para a substituição de colunas da base (levando em conta apenas o fator tempo):

1 Na iteração $k+1$, só executamos uma refatoração da base

a) Se $\phi_k > 1$ e $\phi_k > \phi_{k-1}$ e $k \neq p.v$, $p = 1, 2, 3, \ldots$

b) Ou se $\phi_k > 0.999$ e $k = p.v - 1$, $p = 1, 2, 3, \ldots$[8]

2 Em todos os demais casos damos preferência à atualização.

A memória consumida pelo algoritmo

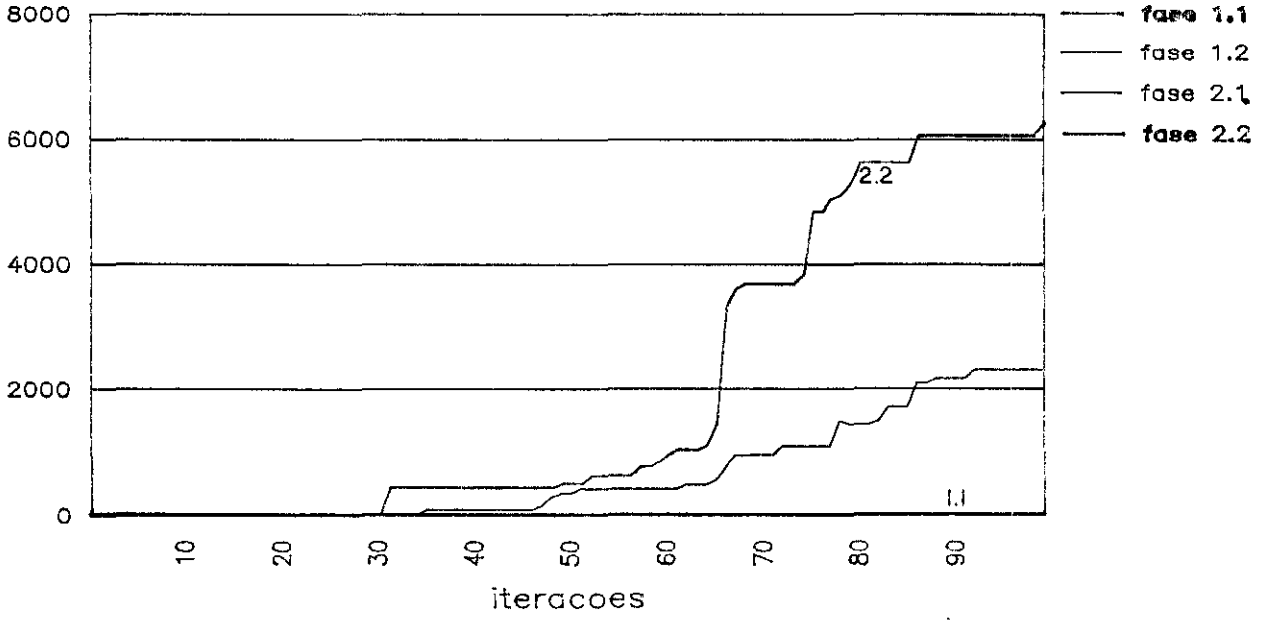
No que tange ao aumento verificado no consumo de memória à medida em que se evita novas fatorações da base, observamos nos gráficos III.7.18 e III.7.19 que durante a fase 1 isto não chega a constituir um problema, pois o aumento de L costuma ser insignificante ou nulo, enquanto o crescimento de U se dá, como sempre, de forma linear e lenta.

---

[8] Isto é feito para que não seja calculada a norma do resíduo.

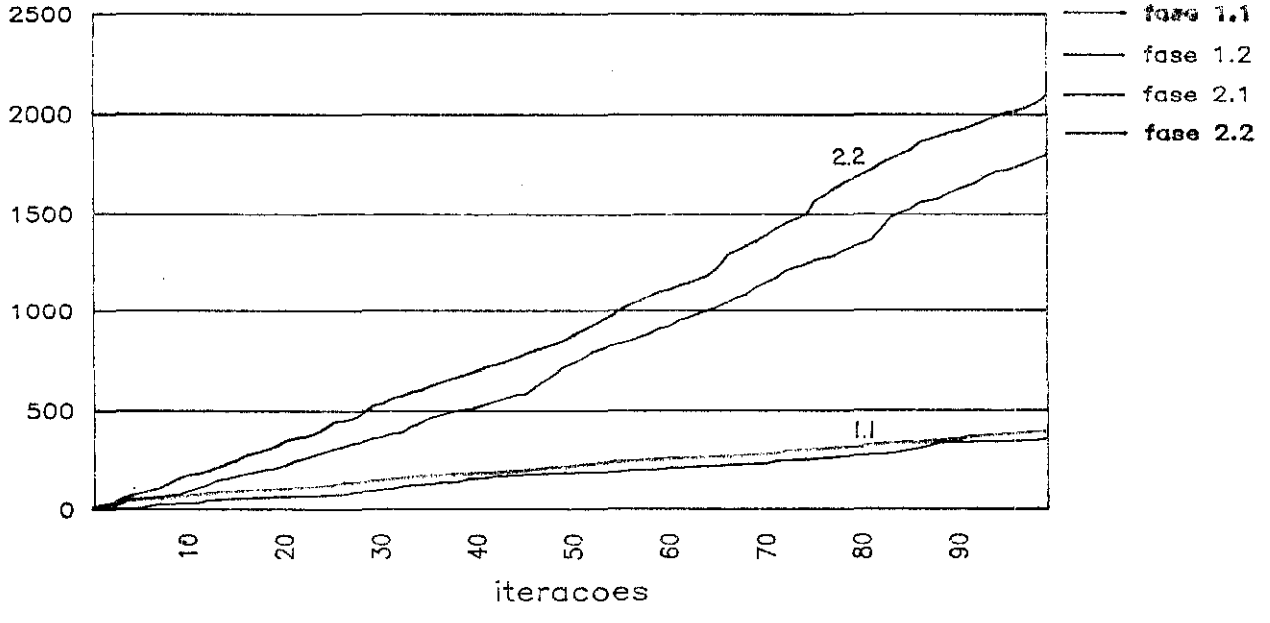
MEMORIA CONSUMIDA POR L NA ATUALIZACAO

Problema com 14 blocos e 2 linhas de acoplamento

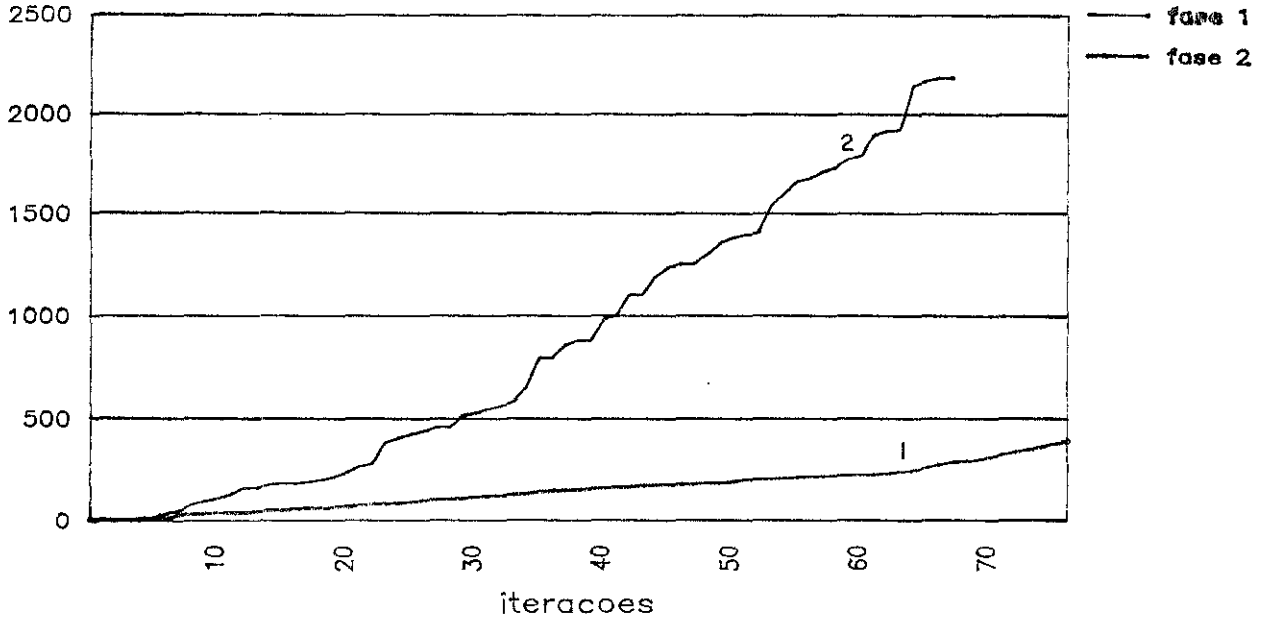


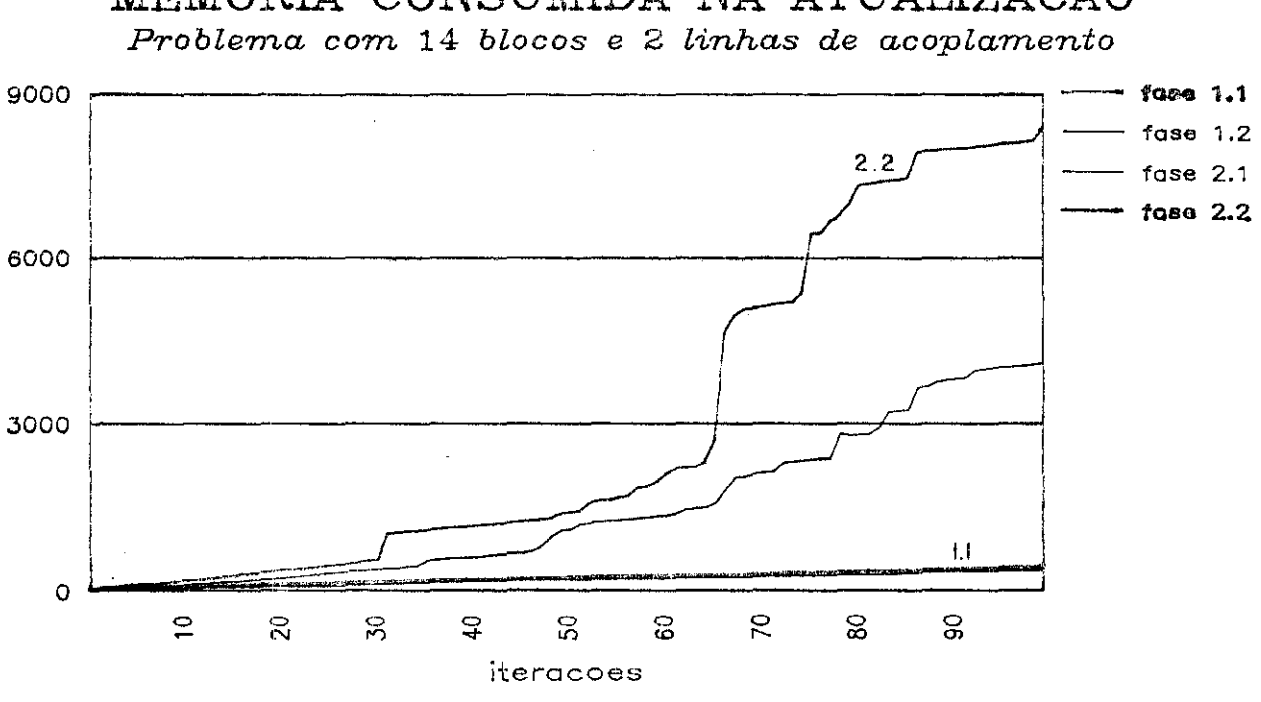
MEMORIA CONSUMIDA POR U NA ATUALIZACAO

Problema com 14 blocos e 2 linhas de acoplamento



- Gráficos III.7.18 e III.7.19 -



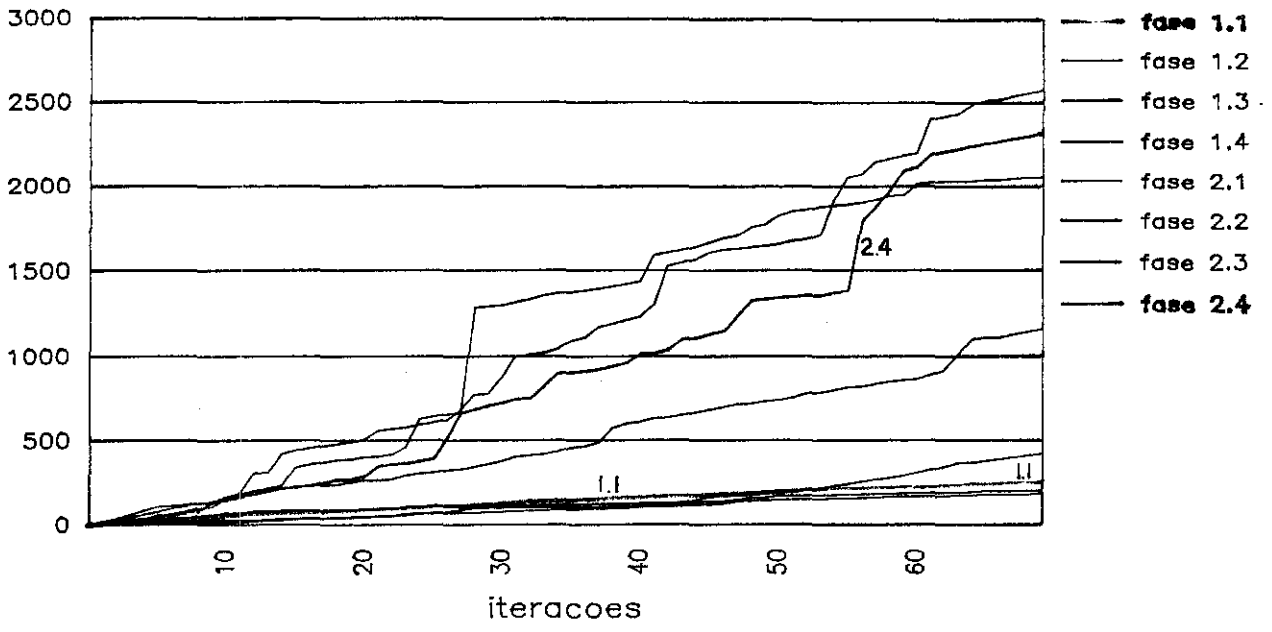


- Gráficos III.7.20 e III.7.21 -

O mesmo não se pode dizer da fase 2, onde o crescimento de L e U é acentuado. Mas este ainda não chega a ser preocupante, pois está, como vimos, fortemente relacionado ao tempo gasto pelo programa, e sabemos que dificilmente matrizes muito grandes induzirão atualizações rápidas. Pode-se inclusive, para verificar que isto realmente ocorre, comparar a memória consumida em vários problemas reais, mostrada nos gráficos III.7.20, III.7.21 e III.7.22, com os gráficos relativos ao fator de economia de tempo a que já nos referimos.

Assim sendo, ainda que a memória disponível ao final da fatoração seja pouca, não seremos forçados, a menos de casos excepcionais, a desistir das atualizações.

O critério utilizado para decidir se uma fatoração é ou não necessária por motivo de espaço é empírico e consiste em verificar se a memória disponível para os elementos criados na atualização é capaz de armazenar ao menos quatro vezes o número de linhas da matriz, caso em que optaremos por esta última.



- Gráfico III. 7. 22 -

CAPÍTULO QUARTO

RESULTADOS NUMÉRICOS

Is there - is there balm in Gilead?
- tell me - tell me, I implore!
Quoth the Raven, "Nevermore".
EDGAR ALLAN POE
(In: The Raven)

Muito já foi dito sobre como melhor aproveitar o método simplex quando se pretende resolver problemas bloco-angulares.

Resta-nos, entretanto, comprovar se toda esta teoria produz, de fato, resultados animadores se comparados aos obtidos por outras propostas há muito sugeridas e já implementadas com algum sucesso.

Passemos, assim, finalmente, à análise qualitativa de nosso algoritmo.

Para avaliar o desempenho do programa recém descrito, resolvemos compará-lo a dois dos algoritmos mencionados no capítulo primeiro, um envolvendo decomposição — o de Rosen — e outro a estratégia das restrições ativas[1], para os quais dispomos de implementações econômicas e eficientes.

Contudo, as versões que possuimos destes programas são dedicadas exclusivamente à formulação de rações, motivo pelo qual seremos forçados a nos restringir a esta classe de problemas.

Mas só decidimos nos submeter a esta aparentemente inconveniente situação na certeza de que, com a boa variedade de modelos oriundos de situações reais[2], estaremos aumentando a confiabilidade dos resultados experimentais obtidos;

Apresentamos, então, no quadro abaixo, um resumo dos problemas utilizados em nossa análise.

---

[1]Para o que contamos com a inestimável colaboração da SOMA - Sistemas, Otimização e Matemática, que nos cedeu, gentilmente, os programas.

[2]Novamente aqui é preciso fazer uma pequena interrupção para agradecer à SOMA que também forneceu os exemplos.

PROBLEMA	1	2	3	4	5	6
DADOS GERAIS						
BLOCOS	4	14	77	48	59	10
LINHAS						
acoplamento	2	2	7	9	4	10
total	58	222	1155	565	704	147
COLUNAS						
originais	104	396	1543	972	994	276
de folga	54	208	1078	469	587	413
total	158	604	2621	1441	1581	689
ELEMENTOS	1492	6264	23114	11351	11917	3822
DADOS DOS BLOCOS						
LINHAS						
mínimo	8	14	9	8	8	8
média	14	16	15	12	12	14
máximo	20	20	20	15	15	20
COLUNAS						
mínimo	24	24	12	14	12	24
média	26	28	20	20	17	28
máximo	30	30	22	23	20	31
ELEMENTOS						
mínimo	192	336	132	126	112	192
média	373	477	300	236	202	382
máximo	600	600	420	322	280	600

Quadro IV.1.1 - Problemas utilizados nas comparações

Em todos os problemas, os blocos correspondem a formulações de rações que são comercializadas no país, assim como as restrições de estoque dos problemas 1, 3, 4 e 5 também se originam de situações reais.

Apenas os modelos 2 e 6 tiveram o acoplamento criado exclusivamente para nossos testes. Mas, mesmo em tais casos, devemos lembrar que, dadas as características desse tipo de restrição, estas poderiam perfeitamente corresponder a problemas reais.

Feitas estas considerações, passemos aos resultados obtidos para cada algoritmo.

SEÇÃO 2 RESULTADOS COMPARATIVOS OBTIDOS

Para que hovesse uma padronização eficaz, capaz de evitar desvios comprometedores nos resultados, todos os programas foram escritos numa mesma linguagem — Turbo Pascal 5.0 — e todas as experiências feitas em um mesmo computador — um compatível com o IBM PC-XT, com processador 8088 (de 4.77MHz) e coprocessador matemático 8087, e 640 Kbytes de memória de rápido acesso.

Pelo mesmo motivo, sempre que possível, optamos por manter uma grande semelhança em alguns pontos cruciais dos programas. Como claro exemplo disto, todos os algoritmos adotam o critério de Dantzig na determinação da coluna que deverá entar na base.

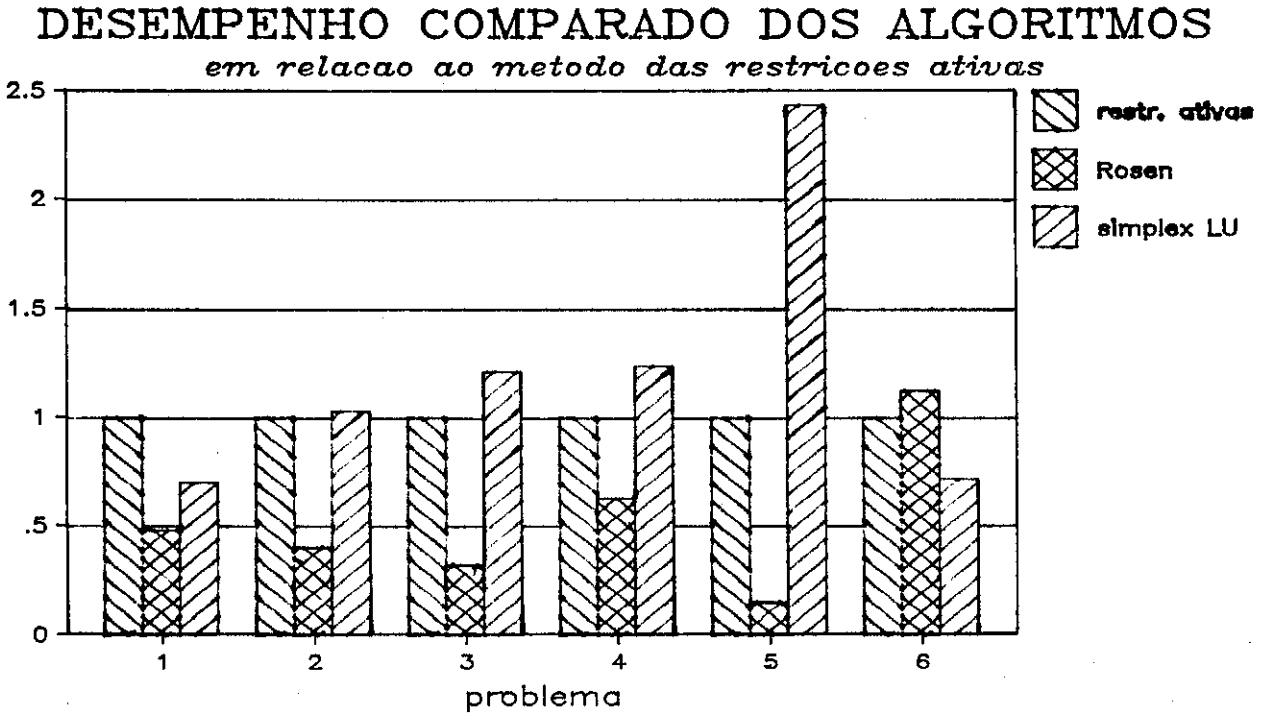
Esperamos ter reduzido, assim, as interferências, muito comuns, provocadas por detalhes que consideramos menos importantes.

Na tabela que se segue exibimos, para cada algoritmo, o tempo gasto na resolução dos problemas (os valores são dados em minutos).

ALGORITMO	RESTRIÇÕES ATIVAS	ROSEN	NOSSO ALGORITMO
PROBLEMA			
1	3.38	1.63	2.37
2	25.27	10.13	25.95
3	589.25	189.92	711.60
4	163.88	102.70	202.88
5	91.98	13.83	224.20
6	30.12	33.78	21.60

Quadro IV.2.1 - Tempo de execução dos diversos algoritmos

A partir dos dados do quadro IV.2.1 foi possível desenhar o gráfico IV.2.1. Nele o tempo consumido por cada método foi dividido pelo valor obtido quando se adota a estratégia das restrições ativas, para formar um índice que é tanto menor quanto mais rápido tiver sido encontrada a solução do problema.



- Gráfico IV. 2. 1 -

Do quadro e do gráfico expostos acima, a única conclusão absolutamente indiscutível que se pode deduzir é que não existe um algoritmo que seja sempre melhor que os outros.

Se fosse necessário defender qualquer um dos métodos, sempre haveria um mínimo de razões capazes de justificar seu uso, embora, de uma forma geral, o algoritmo de Rosen tenha se mostrado sensívelmente melhor na grande maioria dos casos, bem como tenha sido até certo ponto decepcionante o desempenho do programa que estamos apresentando, particularmente no momento em que analisamos o problema de número cinco.

Frente a esta difícil tarefa de comparar o comportamento instável dos métodos (em virtude, inclusive, do pequeno número de experimentos) tentamos identificar alguns fatores capazes de influenciar ou justificar o melhor ou pior desempenho de cada um deles:

r Em primeiro lugar, deve ficar claro que o programa que ora estamos apresentando é o único que não foi elaborado exclusivamente para os problemas de formulação de rações.

Em função de sua generalidade, um bom número de artifícios são por ele evitados, sem que o mesmo aconteça aos demais.

2 Outra relevante diferença existente entre os programas reside no fato do algoritmo de Rosen e o das restrições ativas buscarem a solução ótima resolvendo inicialmente o subproblema composto apenas pelos blocos. Só depois de obtida a solução ótima para este caso, as restrições de estoque são acrescentadas e o problema é resolvido integralmente.

Este procedimento torna estes programas mais rápidos quando aplicados à formulação de rações uma vez que, talvez até pelo pequeno número de restrições de estoque existentes em comparação com o número de blocos, a solução do problema sem acoplamento encontra-se normalmente, suficientemente próxima da solução global.

3 Um aumento da banda de estoque talvez seja suficiente para que este resultado se inverta, pois, como constatamos acima, para um problema com 10 blocos e 10 restrições de estoque, o melhor algoritmo é justamente o que ora apresentamos.

4 Também não devemos deixar de comentar que se obrigássemos o método de restrições ativas a tomar como ponto inicial, o mesmo vértice que utilizamos, muito provavelmente seu

desempenho iria piorar consideravelmente. Isto dar-se-ia porque, além deste algoritmo não incluir uma atualizacão rápida da base (refatorando-a a cada iteração), o procedimento que ele usa para descobrir a variável ou restrição que deixa a base é muito mais complexo e demorado que aquele que descrevemos no capítulo III.

4 Por outro lado, o método que motivou nosso trabalho também possui uma desvantagem que lhe é intrínseca. Seu consumo de memória, que evitamos detalhar aqui, será sempre muito maior que o de um algoritmo que utiliza uma matriz básica reduzida (como o das restrições ativas) ou que emprega decomposição (tal como o de Rosen).

Todos estes comentários fazem com que acreditemos ser possível distinguir grupos de problemas em que algumas das boas características de um ou outro programa induzam seu emprego.

Desta forma, esperamos que o algoritmo que apresentamos seja bastante recomendável quando o número de restrições de estoque for grande, por exemplo, ou quando estivermos interessados em resolver problemas outros que não o de ração, nos quais o acoplamento é menos esparso.

Mesmo para a formulação de rações podemos esperar dele um desempenho satisfatório se os ingredientes com restrição de estoque estiverem presentes na maioria dos blocos.

Em diversos outros casos específicos, os algoritmos de Rosen ou das restrições ativas se comportarão de forma mais conveniente, em função do menor tempo ou mesmo da pouca memória consumida por eles.

CAPÍTULO QUINTO

UMA TENTATIVA DE ACELERAÇÃO

Em uma parcela considerável dos problemas bloco-angulares, o número de restrições de acoplamento é pequeno em comparação ao total de linhas da matriz tecnológica. Este é o caso, por exemplo, dos problemas de formulação de rações, onde há, normalmente, um grande numero de blocos (ou seja, rações), mas poucos ingredientes com estoque limitado.

Como conseqüência disto, no método simplex, a maior parte das atualizações da matriz básica é feita substituindo-se colunas de um mesmo bloco.

Parece, assim, a primeira vista, antieconômico executar, neste caso, a cada passo do simplex, inúmeras operações que envolvem blocos outros que não aquele que participa diretamente da mudança de base.

Procuremos, então, imaginar uma nova estratégia de atualização e de resolução de sistemas que seja capaz de explorar esta característica em proveito do tempo de execução do programa. Com menos ênfase, estabeleçamos a diminuição da memória exigida pelo programa e a melhoria da estabilidade da solução como objetivos secundários, uma vez que não constituem uma deficiência de nosso algoritmo.

SEÇÃO 1 INVESTIGAÇÕES PRELIMINARES

Imaginemos, inicialmente, que a coluna que entra e a coluna que sai da base pertencem ao último bloco da matriz tecnológica.

Analisemos, agora, as conseqüências disto sobre

A refatoração da base

Num caso como este, é fácil perceber que, se precisarmos refatorar a base, bastará fazê-lo na porção da matriz que corresponde às colunas do último bloco e às do acoplamento, como vemos em destaque no exemplo da figura V.1.1:

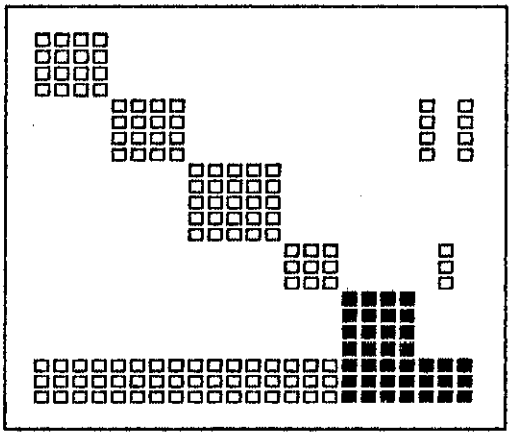


Fig V.1.1 - A base a ser refatorada

Isto decorre do fato de que, a cada passo da fatoração, eliminamos, da submatriz que ainda resta a ser decomposta, uma linha e uma coluna. Donde, se apenas o último bloco da base tiver sido modificado, as linhas e colunas correspondentes ao penúltimo bloco e todos os anteriores não sofrerão alteração.

A atualização da base

De forma análoga, se fosse necessário, no mesmo caso, atualizar a base ao invés de refatorá-la, não modificaríamos nenhum outro bloco que antecede ao último, bem como a parcela do acoplamento a eles correspondente, tornando também este processo bastante rápido.

Mas, não precisamos parar por aí. Afinal, se nos é dado o poder de imaginar, façamo-lo da forma que nos seja mais conveniente. Consideremos, então, que entre duas refatorações sucessivas, todas as atualizações envolverão apenas colunas do último bloco ou colunas das variáveis de folga do estoque.

Isto evitaria que sequer um elemento não nulo fosse criado durante este processo, fazendo com que a matriz L da base decomposta passe a possuir sempre elementos em número exatamente igual ao obtido quando da fatoração. Consegue-se, assim, uma economia de memória nem tão desprezível, ao que se acrescenta a vantagem de ser possível eliminar o termo limitante μ utilizado nos pivoteamentos (vide equação II.4.33), pois que a razão deste existir se restringia à redução do número de componentes de L e U oriundos da atualização.

E, como se não houvessem tantas vantagens, resta mencionar, ainda, que não seria mais necessário manter uma estrutura de dados específica para a atualização, facilitando este processo e economizando ainda mais tempo.

SEÇÃO 2 DESCRIÇÃO DO NOVO ALGORITMO

Expostos desta forma os fatos, seria bastante razoável tentar induzir que a atualização da base no algoritmo simplex se dê apenas entre colunas do último bloco e aquelas correspondentes a variáveis de folga do estoque. Resta, contudo, imaginar, ainda, o que fazer para que isto ocorra.

Um novo teste de otimalidade

A forma mais fácil de conseguir que a coluna que entra na base pertença ao último bloco é procurar primeiro lá. Ou seja, passar a percorrer o vetor $\bar{c}^N$, dado pela equação

$$\bar{c}^N = c^N - \lambda^T.A^N, \qquad (V.2.1)$$

a partir da última posição, adotando um dos seguintes critérios para a determinação da coluna:

1 Escolher sempre a última candidata, isto é, a primeira que encontrarmos, já que estamos percorrendo o vetor $\bar{c}^N$ de trás para frente;

2 Escolher, como no item anterior, a última candidata, *i*, sujeitando-a, entretanto, a uma tolerância ν, de forma a rejeitá-la se $|\bar{c}^N_i| < \nu$, a menos que nenhuma delas satisfaça a este critério, caso em que se escolhe a de maior valor absoluto;

3 Escolher a melhor candidata do último bloco segundo o critério de Dantzig (ou seja, aquela que possui maior valor absoluto). Caso não exista nenhuma, passar para o bloco anterior e assim sucessivamente.

4 Unir os itens *2* e *3* acima, o que significa empregar o critério de Dantzig por blocos, mas com filtro.

Em qualquer destas hipóteses, será necessário resolver, primeiro, o sistema

$$B^T.\lambda = c_B \qquad (V.2.2)$$

Para tanto, após permutar os elementos de c_B, resolvemos inicialmente

$$U^T.w = c_B \qquad (V.2.3)$$

e então

$$L^T.\lambda = w \qquad (V.2.4)$$

Para economizar tempo, este último sistema, no qual a matriz L^T é percorrida de baixo para cima, pode ser resolvido por partes, sujeitando cada componente encontrada em λ aos testes dos itens *1* e *2* acima, ou calculando este vetor bloco a bloco a partir do último, parando quando as condições *3* ou *4* forem satisfeitas.

Também é possível reduzir o tempo de solução do sistema V.2.3 armazenando, entre iterações sucessivas, os elementos de w referentes aos blocos que antecedem o último, pois estes não sofrem alteração se a atualização se der apenas entre colunas do bloco final.

Acelerando o teste da razão

Por sua vez, o processo de determinação da coluna que irá sair da base exige que encontremos o vetor y dado por

$$B.y = A_s \qquad (V.2.5)$$

Neste caso, resolvemos

$$L.w = A_s \qquad (V.2.6)$$

e, em seguida,

$$U.z = w \qquad (V.2.7)$$

No sistema V.2.6, supondo que a coluna A_s pertença ao i-ésimo bloco da base (normalmente o último, se usarmos o teste de otimalidade recém descrito), os elementos deste vetor que pertencem às linhas correspondentes a todos os demais blocos serão nulos. Conseqüentemente, as posições equivalentes do vetor w também o serão, sendo desnecessário calculá-las.

A figura V.2.1 abaixo ilustra com fidelidade a diferença entre o sistema que iríamos resolver normalmente, envolvendo toda a matriz L, e o que resolvemos de fato, aquele bem menor que aparece em destaque.

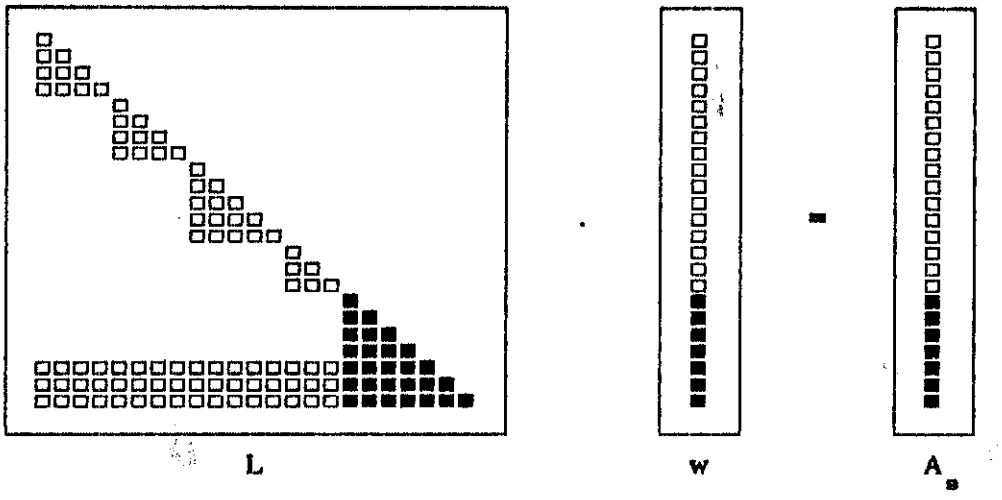


Fig V.2.1 - O sistema $L.v = A_s$

De forma semelhante, para o sistema V.2.7, nem todas as posições de z serão calculadas. Os elementos de z correspondentes aos blocos quadrados da base, exceto aquele que contém A_s, se for este o caso, podem ser ignorados, pois serão nulos. Com isso, nosso sistema se resumirá aos blocos não quadrados e ao i-ésimo bloco (geralmente o último), donde calcularíamos, por exemplo, apenas a parte de z destacada na figura V.2.2:

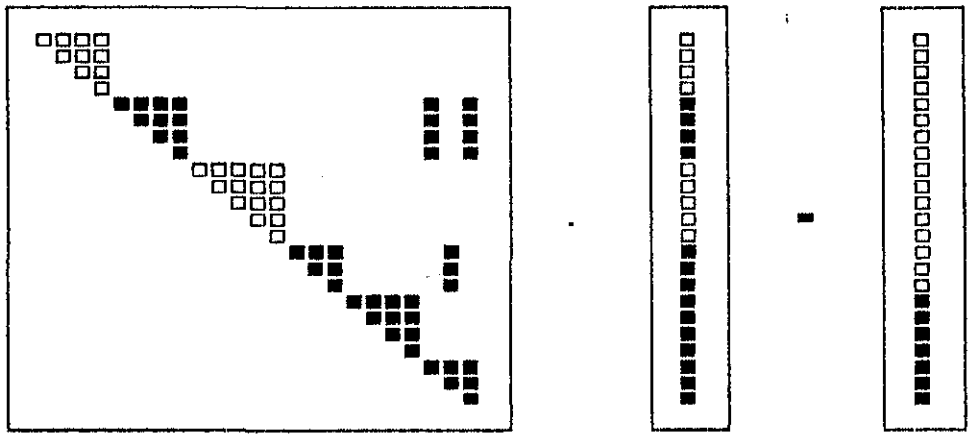


Fig V.2.2 - O sistema U.z = w

Quanto às permutações dos blocos quadrados, excetuando-se eventualmente o bloco ao qual pertence A_s, estas também podem ser evitadas, completando o processo de resolução do sistema V.2.5.

O que fazer se as coisas não se comportarem como queríamos

Até agora estivemos tratando de uma situação ideal, onde a atualização da base se consumava pela substituição de colunas do último bloco da matriz. Porém, verdadeiramente, nem sempre isto ocorrerá e é preciso estar preparado para tal adversidade.

Como o algoritmo de atualização da base foi modificado para só aceitar alterações no último bloco e em uma pequena parcela do acoplamento, será preciso refatorar a base sempre que a coluna que entra ou a que sai da base pertencer a outro bloco.

Mas, embora este processo seja muito mais demorado que a atualização, nem tudo está perdido.

Suponhamos, por exemplo, que a coluna que entra na base pertence a um bloco qualquer, k. Neste caso, não somos obrigados a refatorar toda a base, mas apenas a parcela correspondente aos blocos k, ..., n, o que pode reduzir consideravelmente o trabalho, principalmente se nos lembramos de que procuramos esta coluna seguindo o vetor $\overline{c}$ de trás para frente.

Tendo em mente, ainda, que *o número de linhas de acoplamento é pequeno e a maior parte das substituições de colunas da base se dá dentro de um mesmo bloco*, podemos aproveitar o fato de que a ordem dos blocos do problema é estritamente arbitrária para alterá-la segundo a nossa conveniência. Assim, também pode ser considerada muito interessante a idéia de permutar os blocos de forma que aquele que contém a coluna que entra na base passe a ser o último. Com isso podemos garantir que a maior parte das mudanças de colunas básicas vá se dar no último bloco do problema.

Agindo desta forma, conseguiremos atender, na maior parte das iterações do simplex, às exigências feitas no início desta seção. Nas pouquíssimas vezes em que isso não for possível será preciso ter paciência e recordar que o pequeno aumento do tempo numa tal situação deve ser compensado, com muita folga, pelas inúmeras iterações em que o novo algoritmo é mais rápido que o original.

SEÇÃO 3 ALTERAÇÕES NECESSÁRIAS

Se fossemos comentar pormenorizadamente todas as subrotinas que precisam ser modificadas no programa original para atender ao que foi proposto na seção anterior, gastaríamos mais espaço do que aquele consumido pelo capítulo terceiro na íntegra.

Assim sendo, melhor será poupar o leitor de mais uma fastidiosa leitura e abordar a matéria de forma mais objetiva, remetendo os interessados por maiores detalhes ao apêndice C, que contém a listagem do algoritmo deste capítulo.

Estrutura de dados

Com a necessidade de permutarmos freqüentemente os blocos, tornou-se indispensável remodelar completamente a estrutura de dados para torná-la menos rígida.

Para tanto, extinguimos a maior parte dos vetores que, no algoritmo anteriormente apresentado, se referiam ao problema de uma forma global. Estes deram lugar a um único vetor denominado *matriz*, composto de registros que contêm todas as informações separadas bloco a bloco.

Assim, para o bloco *i* da matriz, o registro correspondente, *matriz[i]*, contém, dentre outros, os seguintes campos:

1 *ncor* e *nlin*, respectivamente, número de colunas e linhas do problema na forma mista;

2 *ncca*, número de variáveis de folga correspondentes a restrições canalizadas;

3 *ncfo*, número de variáveis de folga ou excesso de restrições não canalizadas;

4 *ncar*, número de variáveis artificiais;

5 *ncol*, número de variáveis que pertencem à base atual;

6 *dad*, vetor que contém a matriz tecnológica do bloco;

7 *lest*, vetor que armazena, do bloco *i* da matriz L, a parte referente ao acoplamento;

8 Além dos vetores *afart*, *fobas*, *lim*, *obj2*, *b* e *lsup*, que contêm as mesmas informações dos seus homônimos do programa do capítulo III, mas referem-se apenas ao bloco *i*;

E uma vez que esperamos ver o desempenho deste novo algoritmo comparar-se aqueles elaborados especialmente para formulação de rações, também tivemos a preocupação de exigir que a estrutura de dados fosse melhor adaptada a este tipo de problema, economizando memória[1]. Por isso, para o mesmo bloco *i* da matriz tecnológica, as informações referentes ao acoplamento são armazenadas através de duas outras componentes de *matriz[i]*, a variável *prod* e o vetor *est*, onde:

9 *est[k]* indica qual coluna do bloco possui um elemento não nulo na *k*-ésima linha de acoplamento[2]; e

10 *prod* fornece o valor de todos os elementos não nulos da parte do acoplamento referente ao bloco em questão[3].

---

[1]Desta forma será preciso submeter o programa a ligeiras alterações para vê-lo resolver problemas de outra natureza.

[2]Lembremo-nos que, nos problemas de rações, não existe, em cada bloco, mais de uma coluna relacionada a uma mesma restrição de estoque

[3]Estes elementos são todos iguais. Mais uma característica dos problemas de ração.

Outro aspecto importante do problema de formulação de rações é o fato, comum a outros modelos, do acoplamento não possuir colunas "originais".

Tambem preferimos tomar partido desta característica, o que, por um lado simplifica nosso algoritmo, mas restringe ainda mais o tipo de aplicação a que ele se destina (ainda que não seja trabalhoso modificá-lo).

A ordem dos blocos

Esta nova estrutura de dados torna simples a tarefa de permutar os blocos quando necessário.

Com todas as informações divididas convenientemente e definindo um único novo vetor, *indblo*, que armazena a ordem em que se encontram os blocos do problema original, jamais será necessário efetuar uma permutação "física" dos dados, bastando para tanto alterar algumas componentes de *indblo*[4].

Exemplificando, suponhamos que pretendemos trocar todas as informações relativas aos blocos 5 e 9 de lugar. Neste caso, basta fazer com que *indblo[5]* assuma o valor armazenado em *indblo[9]* e vice-versa, lembrando que, se *indblo[5]* = 2,

---

[4]A única exceção é vetor *cabas*, que ainda é global e, portanto, precisa ser modificado a cada permutação de blocos.

queremos dizer que o segundo bloco do problema original ocupa, no momento, a quinta posição.

Exemplo numérico

Para tornar mais clara a nova estrutura de dados, vejamos então como é armazenado o problema da fig. V.3.1 abaixo.

$$
\begin{array}{lrrrrrrrrcr}
\text{minimizar} & -2x_1 & -1x_2 & -1x_3 & -4x_4 & -1x_5 & -6x_6 & -2x_7 & -5x_8 & & \\
\text{sujeito a} & +1x_1 & +2x_2 & +5x_3 & -3x_4 & & & & & \leq & 18 \\
 & +2x_1 & -1x_2 & +7x_3 & +2x_4 & & & & & \geq & 12 \\
 & +1x_1 & +1x_2 & +1x_3 & +1x_4 & & & & & = & 1 \\
 & & & & & +6x_5 & -1x_6 & -3x_7 & +2x_8 & \leq & 15 \\
 & & & & & +4x_5 & -7x_6 & +2x_7 & +9x_8 & \geq & 7 \\
 & & & & & +1x_5 & +1x_6 & +1x_7 & +1x_8 & = & 1 \\
 & 10x_1 & & & & +6x_5 & & & & \leq & 30 \\
 & & 10x_2 & & & & & & +6x_8 & \leq & 42 \\
 & & & & 10x_4 & & +6x_6 & & & \leq & 9
\end{array}
$$

e

$$
\begin{array}{llll}
x_1 \leq 6 & x_3 \leq 8 & x_5 \leq 7 & x_7 \leq 4 \\
x_2 \leq 4 & x_4 \leq 3 & x_6 \leq 2 & x_8 \leq 5
\end{array}
$$

Fig. V.3.1 - Um problema bloco-angular semelhante ao encontrado em formulação de rações

Deste problema, seu segundo bloco é guardado no registro *matriz[2]*, cujas componentes que por ora nos interessam estão expostas na figura V.3.2[5].

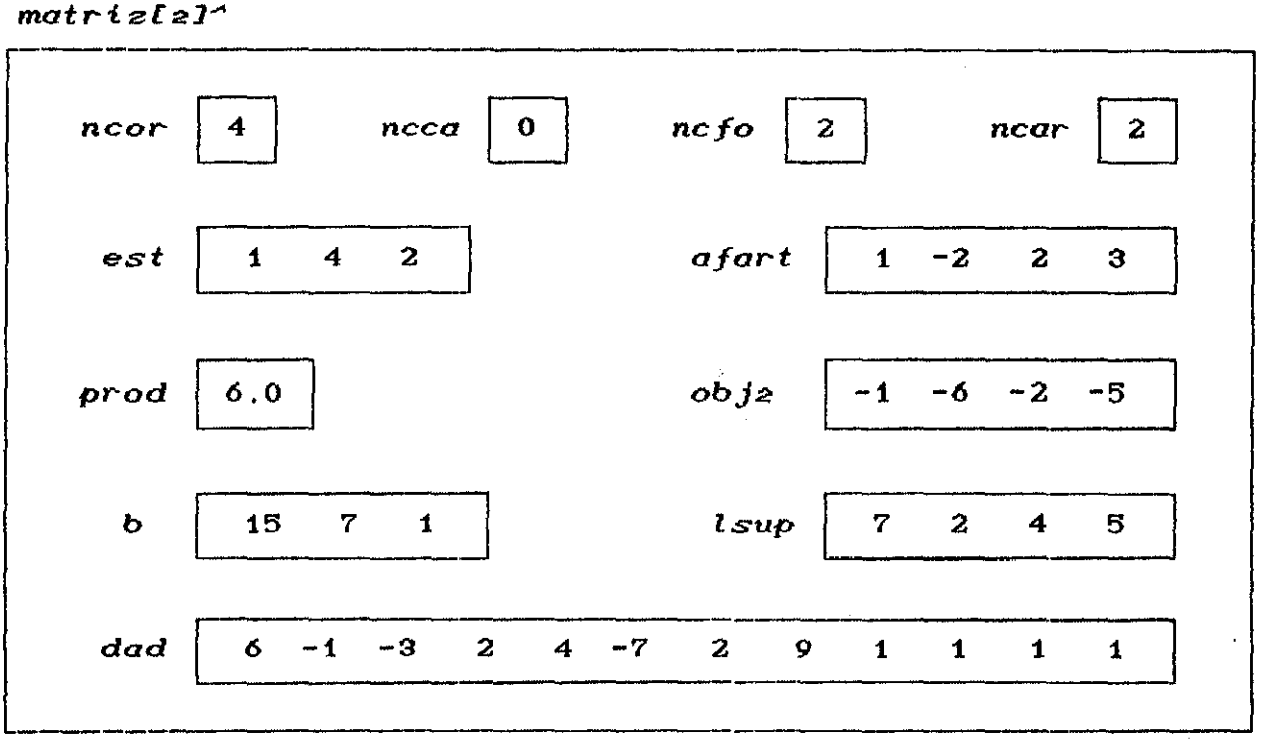


Fig. V.3.2 - Algumas componentes do registro matriz[2]

[5]Algumas componentes deste registro foram suprimidas por serem muito semelhantes àquelas já expostas no capítulo III.

Fatoração

Como dissemos, todas as subrotinas do programa foram mais ou menos modificadas para que se adaptassem ao novo algoritmo.

Desconsiderando o que se deve exclusivamente à nova estrutura de dados, comentemos as mudanças mais importantes, começando pela fatoração da base que, neste novo programa, é feita sempre que:

1 As sucessivas atualizações do último bloco da matriz tenham provocado um resíduo no cálculo do vetor solução que é maior que o permitido; ou

2 A coluna que entra ou que sai da base não pertença ao último bloco mas a algum outro, cujo índice denotaremos por *k*.

Em ambos os casos não é necessário refatorar toda a matriz. Se ocorre o que foi descrito no item *1*, apenas o último bloco da base deve ser recalculado, além, é claro, da sub-matriz composta pelas interseção das colunas e linhas do acoplamento. Se o motivo é aquele exposto no item *2*, os elementos de *L* e *U* pertencentes aos blocos que antecedem a *k* permanecem os mesmos.

Evitamos, desta forma, perder tempo para obter dados de que já dispomos, não mais que a obrigação de todo otimizador.

Atualização

O procedimento **Atulizaçao**, por sua vez, foi bastante simplificado, já que ele só será utilizado quando a coluna que entra e a que sai da base pertencerem ao mesmo bloco.

Com a eliminação da complexa estrutura de dados anteriormente empregada para armazenar L, várias partes desta subrotina foram também eliminadas. Apesar disto, seu conteúdo nenhuma alteração sofreu.

QuemEntra

O procedimento **QuemEntra** foi modificado para permitir que o teste de otimalidade do simplex seja feito percorrendo o vetor $\overline{c}^N$ (eq. V.2.1) a partir de seu final e por blocos.

A busca da coluna que entra na base é dada por concluída no momento em que, num bloco qualquer, a melhor candidata atender a um critério de tolerância (segundo o item *4*, pag. 188).

Resolução de Sistemas

Todos os procedimentos ligados a resolução de sistemas sofreram alterações significativas, que passamos a descrever.

1 O sistema LtZX (requisitado pelo procedimento **QuemEntra**, recém comentado) é, agora, resolvido por partes. Inicialmente, apenas a parcela correspondente ao acoplamento e ao último bloco do vetor z é calculada. As componentes de z referentes a outros blocos só são determinadas quando e se isto for necessário.

2 Nas vezes em que a atualização da base se dá tão somente entre colunas do último bloco, a maior parte do vetor c_B nã se modifica, donde armazenamos o vetor w do sistema $U^T w = c_B$ (eq. V.2.3), para que o procedimento UtXY possa reaproveitá-lo em iterações posteriores.

3 Da mesma forma, como o novo procedimento **Atualização** envolve apenas o último bloco da base, sempre que é requisitada por ele, a rotina UtXY só precisa considerar os elementos pertencentes a este bloco.

4 Uma vez que a coluna que entra na base, A_s, só contém elementos não nulos nas linhas correspondentes a um determinado

bloco, as demais componentes deste vetor são ignoradas ao resolvermos o sistema L.w = A_s. Pelo mesmo motivo, sempre evitamos manipular os elementos de w que são iguais a zero.

5 Quando a atualização da base envolve apenas o último bloco desta matriz, somente este bloco e aqueles que não são quadrados, alem do acoplamento, irão produzir elementos não nulos no vetor z, solução de U.z = w, donde os demais blocos também são ignorados.

Subrotinas Calc_Z e CalcNorma

Como tantos outros procedimentos, **Calc_Z** e **CalcNorma** foram re-escritos para evitar que calculemos aquelas componentes que não foram alteradas desde a última fatoração.

Procedimento OrdenaCabas

Embora o processo de ordenação dos elementos do vetor *cabas* que utilizamos seja muito rápido se os elementos já estão em ordem, é mais elegante evitar que componentes já ordenadas sejam analisadas desnecessariamente.

Assim, só reordenamos o vetor *cabas* a partir da posição do primeiro elemento fora de ordem, que, no nosso caso, poderá ser um elemento do último bloco, se só atualizamos esta parte da base, ou de um bloco anterior, situação em que apenas este elemento estará fora de ordem noutro bloco que não o último.

Procedimento MudaOrdem

Uma única rotina nova foi criada especialmente para este algoritmo "acelerado". Trata-se do procedimento **MudaOrdem**, responsável pela permutação dos blocos.

Supondo que o último bloco será permutado com o de índice *k*, podemos descrever este procedimento com o auxílio de seu *pseudo-code*:

1. Para cada bloco da base, a partir daquele cujo índice é *k*
    1. Atualizar o vetor *cabas*
2. Permutar *indblo[k]* com *indblo[nbloc]*.

Finalmente, com este novo algoritmo, sentimo-nos muito mais a vontade para comparar programas de formulações de rações.

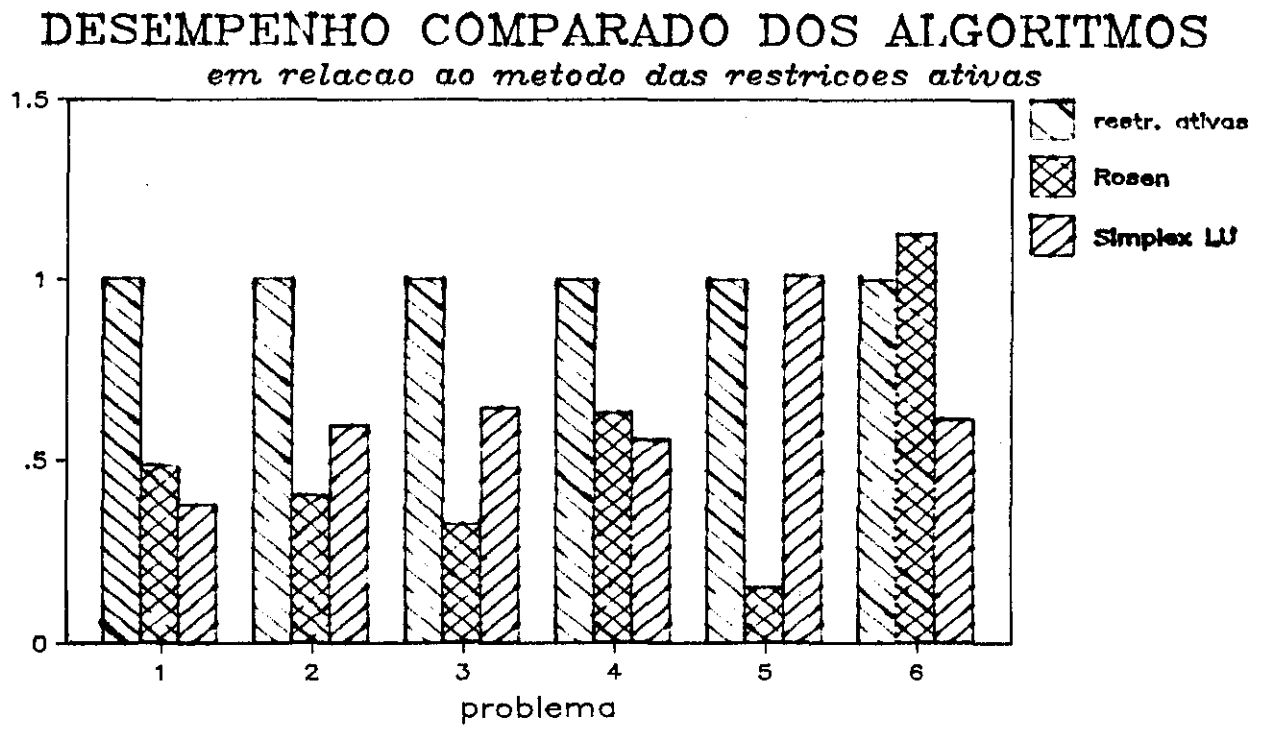
Se no capítulo passado nos encontrávamos às voltas com as conseqüências do conhecimento prévio do problema ao qual o programa se destina sobre seu possível desempenho, agora não há mais desculpas possíveis, pois passamos a considerar a maior parte das idiossincrasias, se assim podemos dizer, da classe de problemas que estamos estudando.

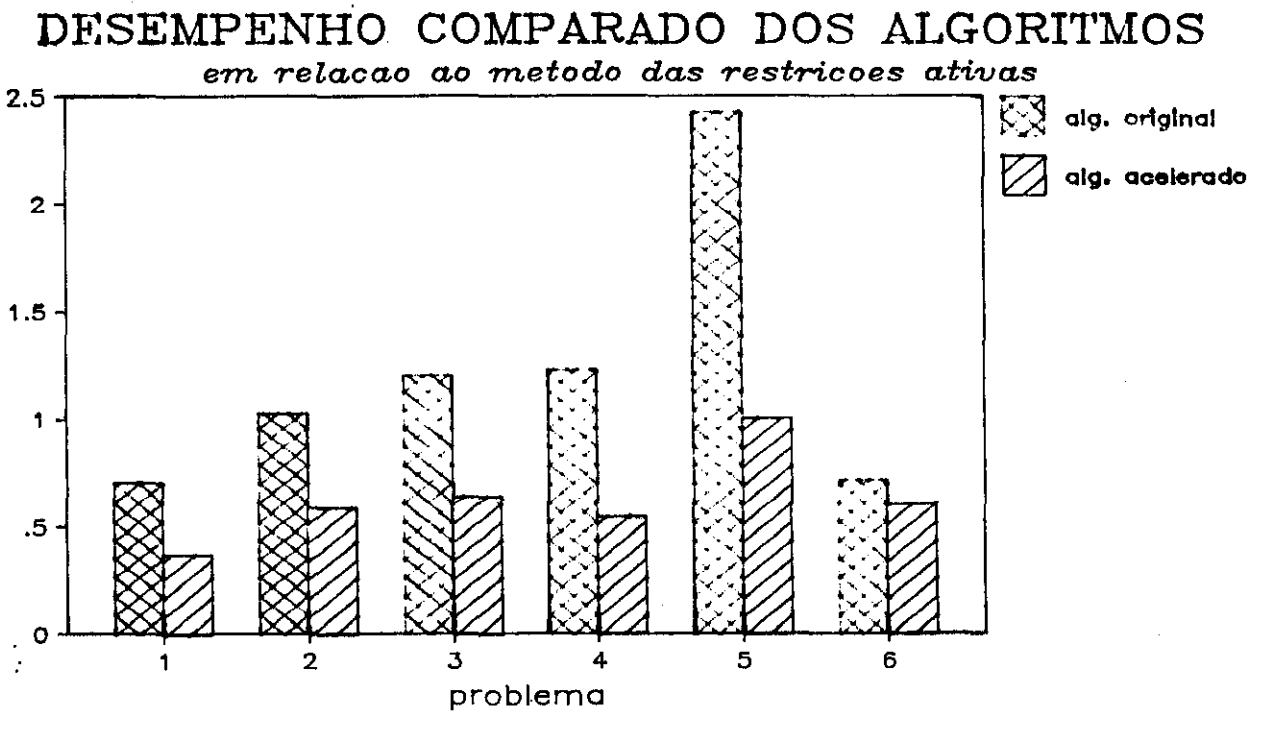
Lembremo-nos, então, da tabela IV.2.1 e comparemos os resultados já obtidos e os tempos (em minutos) relativos ao algoritmo "acelerado", dados abaixo.

ALGORITMO	RESTRIÇÕES ATIVAS	ROSEN	SIMPLEX LU ACELERADO
PROBLEMA			
1	3.38	1.63	1.27
2	25.27	10.13	15.00
3	589.25	189.92	378.45
4	163.88	102.70	90.53
5	91.98	13.83	92.92
6	30.12	33.78	18.37

Quadro V.4.1 - Tempo de execução dos diversos algoritmos

Com os dados deste quadro e também conhecendo o desempenho do algoritmo do capítulo III, elaboramos os gráficos V.4.1 e V.4.2 que, como aquele do capítulo anterior, fornece índices que são tanto menores quanto menos tempo for consumido pelo algoritmo na resolução dos problemas.





- Gráficos V.4.1 e V.4.2 -

Observando o gráfico V.4.2, verifica-se facilmente que o novo algoritmo está muito mais adaptado ao tipo de problema que estamos abordando, pois em nenhum momento o programa do capítulo III foi melhor que este que apresentamos agora.

Também em relação aos outros algoritmos, esta nova versão teve um desempenho exemplar, normalmente consumindo menos tempo que o método das restrições ativas e não poucas vezes suplantando o algoritmo de Rosen.

Por outro lado, o número de iterações executadas na tentativa de se obter uma solução ótima foi razoavelmente maior neste último método que em todas as outras alternativas testadas. Entretanto, isto foi compensado com folga, como já previíamos, pelo reduzido tempo consumido em cada iteração.

O grande consumo relativo de memória (embora o tenhamos reduzido bastante) continua, então, infelizmente, a ser o único inconveniente que se manteve em todas as tentativas de se implementar um algoritmo simplex com decomposição LU.

Mesmo assim são poucos os casos de formulação de rações em que não poderíamos aplicar este novo programa.

CAPÍTULO SEXTO

CONCLUSÕES

Tudo que faço ou medito
Fica sempre na metade.
Querendo, quero o infinito.
Fazendo, nada é verdade.

FERNANDO PESSOA
(in: Cancioneiro)

O objetivo principal deste trabalho era o de apresentar e analisar o desempenho de um algoritmo que, baseado no método simplex, decompusesse a base em matrizes triangulares e que fosse devotado a problemas com restrições na forma bloco-angular.

Elaborados os programas, fica o sentimento de que este objetivo foi alcançado e de que embora com algumas restrições, os algoritmos que propusemos podem ser implantados imediatamente para solucionar problemas desta natureza.

Fazemos, é claro, a ressalva de que, normalmente, para problemas comuns de formulação de rações pode não ser aconselhável empregar a primeira versão apresentada para o simplex com decomposição LU. Mas o algoritmo "acelerado" do capítulo V aparenta ser uma alternativa competitiva, em que pese o pequeno número de experimentos que fizemos e a impossibilidade atual de isolar com alguma confiabilidade aquelas características que, existindo num problema, podem favorecer um ou outro método.

Também devemos dizer que é possível, ainda, melhorar o desempenho dos programas se alterarmos algumas das inúmeras tolerâncias que, propositalmente, o leitor só irá conhecer se estiver disposto a, pacientemente, ler as listagens que se encontram no apêndice C.

Novos valores para estas tolerâncias certamente não irão produzir resultados espantosos, mas sempre poderão tornar o programa um tanto mais rápido, econômico ou estável.

Uma surpresa que nos pareceu interessante e que deixamos para relatar neste capítulo é o ótimo desempenho do algoritmo "acelerado" durante a fase I do simplex. Em todos os problemas testados foi impressionante a rapidez com que este programa obtinha uma solução factível, consumindo para isto apenas uma fração bastante reduzida do tempo global.

Este resultado é bastante razoável se tivermos em conta que, para eliminar as variáveis artificiais de um determinado bloco, normalmente usa-se colunas do mesmo bloco, donde pouquíssimas são as iterações que, nesta fase, não se enquadram no que consideramos ideal para este algoritmo.

Um último comentário que resta ser feito refere-se ao fato de que a maioria dos blocos nos problemas de rações tem um grau de esparsidade elevado, que pode ser levado em conta para reduzir, talvez, ainda mais o tempo de execução do programa (talvez até pela divisão de cada bloco em blocos menores).

Nossa expectativa agora, leitor, passa a se concentrar nas surpresas que nos reservam os especialistas em métodos de ponto interior.

APÊNDICE A

DUAS OU TRÊS COISAS SOBRE PASCAL

Para os leitores não habituados à linguagem PASCAL, vamos expor agora, muito resumidamente, aquilo que dela julgamos ser indispensável conhecer para uma melhor compreensão do capítulo terceiro e dos apêndices B e C deste texto, incluindo alguns tipos de dados que não são comuns às outras linguagens e os operadores e comandos mais importantes utilizados.

Dos diversos tipos de dados existentes em Pascal, muitos irão nos interessar: inteiros, reais, booleanos, arquivos, intervalos, vetores, registros e apontadores, além daqueles tipos que são definidos pelo próprio usuário. Destes, os quatro primeiros o leitor já deve conhecer, motivo pelo qual nos absteremos de fazer maiores comentários.

Iremos apresentar, então, uma noção dos últimos cinco tipos, com a brevidade e superficialidade que a conveniência exige.

Tipos de dados definidos pelo usuário

Ao programador em Pascal insatisfeito com os números inteiros, a linguagem faculta a criação de seus proprios tipos de dados escalares[1], desde que cada valor seja representado por um identificador diferente.

[1]Aqueles que podem ser ordenados e enumerados.

O termo abaixo representa, por exemplo, um tipo de dado denominado *semana*.

semana = (segunda,terça,quarta,quinta,sexta,sabado,domingo)

E se definimos *dia* como uma variável do tipo *semana*, ela poderá, então, assumir qualquer um dos valores que aparecem entre parêntesis.

Intervalos

Também são considerados tipos de dados os intervalos criados a partir de outros tipos definidos anteriormente, como os exemplos abaixo:

-100..100 — intervalo do tipo inteiro;
quarta..sábado — intervalo do tipo *semana*.

Vetores

Os Vetores, presentes em quase todas as linguagens, têm seus elementos representados em Pascal por um identificador qualquer, seguido de um índice entre colchetes.

Assim, o termo *ncol[5]* se refere à quinta componente do vetor *ncol*.

Registros

Os registros são estruturas mais complexas, englobando várias informações diferentes mas correlacionadas, representadas através de um número fixo de componentes denominados *campos*, cada um destes pertencente a um determinado tipo de dado. Podemos imaginar, por exemplo, uma matriz cujos elementos, esparsos, são armazenados num único vetor chamado *matriz*, que contém registros do tipo:

```
elemento = record
             linha,coluna: integer;
             valor       : real;
           end
```

Cada registro armazena as informações necessárias à identificação de um determinado elemento da matriz, incluindo a linha e a coluna a qual ele pertence e o seu valor.

Os campos de um registro são referenciados através do identificador composto pelo nome do registro seguido de um ponto e do nome do campo. O termo *matriz[2].linha* corresponde, então, ao campo *linha* da segunda componente do vetor *matriz*.

Apontadores

É possível ainda, em Pascal, gerar e eliminar certas variáveis durante a execução de um programa. Tais variáveis são denominadas *dinâmicas*, em oposição às *estáticas*, declaradas normalmente e com espaço reservado durante a compilação.

Cada variável dinâmica, exatamente por não ser declarada inicialmente, só pode ser referenciada através de um *apontador*, que armazena a sua posição na memória do computador.

Imaginemos, para tornar as coisas mais claras, que estamos trabalhando com registros como o que é mostrado abaixo:

```
info = record
         x,y: real;
       end
```

Para manter uma lista ligada de registros deste tipo, criamos em cada um deles um novo campo que indica onde está situado o registro seguinte. Este campo chamaremos *proximo*, como vemos a seguir:

```
ligacao = ^info;

info    = record
            x,y    : real;
            proximo: ligacao;
          end
```

O sinal — ^ — que antecede o termo *info* na primeira linha acima indica que o tipo *ligacao* nada mais é que um apontador, que fornece a posição da memória onde se encontra um dado do tipo *info*. Assim, cada elemento da lista armazenará a posição do seu sucessor, tornando possível a criação de novos elementos toda vez que a lista precisar ser expandida.

O mesmo símbolo — ^ — também é usado para indicar o elemento apontado, donde se *comeco* é uma variável do tipo *ligacao*, o termo *comeco^.x* irá representar a variável *x* do registro cuja posição é fornecida pelo apontador *comeco*.

Além de armazenar estruturas de dados intrinsecamente dinâmicas, como as listas ligadas, os apontadores têm outra aplicação, decorrente da implementação que usamos do Pascal, que é incapaz de endereçar diretamente toda a memória disponível do computador. Em virtude disto, muitas vezes é preciso lançar mão de apontadores apenas para poder utilizar mais memória.

Também é possível, quando pretendemos otimizar problemas diferentes, usar os apontadores para fazer com que os vetores consumam apenas a quantidade mínima indispensável de memória, economizando um espaço normalmente desperdiçado quando as dimensões dos vetores são fixadas antecipadamente pelo programa. Assim, mesmo estruturas de dados estáticas podem ser criadas e eliminadas durante a execução deste.

Os operadores existentes em Pascal são divididos, segundo o tipo de dado a que se aplicam, em *aritméticos*, *relacionais*, *lógicos* e *de conjuntos*. Abaixo, uma lista dos principais deles.

Aritméticos

+	soma
-	subtração
*	multiplicação
/	divisão real
div	divisão inteira
mod	resto da divisão inteira

Relacionais

=	igual a
<>	diferente de
<	menor que
>	maior que
<=	menor ou igual a
>=	maior ou igual a
in	pertencente a (um determinado conjunto)

Logicos

not	negação
or	disjunção
and	conjunção

De conjuntos

+	união
-	diferença
*	interseção

Atribuição

O comando de atribuição é expresso em pascal através do símbolo := (o sinal de dois pontos seguido do de igualdade). O termo

```
raiz := sqrt(x)
```

significa, então, que o valor da expressão *sqrt(x)* será atribuido à variável *raiz*.

Comandos compostos

Chamamos de *comando composto* qualquer conjunto de comandos que devem ser executados numa seqüência determinada e que pode ser tratado como um único corpo.

Um comando composto aparecerá sempre entre os símbolos begin e end, como no exemplo:

```
begin
  raiz := sqrt(x);
  write(x,'   ',raiz);
end;
```

Comandos repetitivos

São comandos repetitivos em pascal: *while*, *repeat* e *for*. A forma correta de expressar cada um deles é dada abaixo.

while <expressão booleana> do <comando>

repeat <conjunto de comandos> until <expressão booleana>

for <variável> := <valor inicial> to <final> do <comando>

Na primeira forma, o termo <comando> (que pode ser um comando composto) será executado até que a expressão booleana seja falsa. Se o valor desta já é falso inicialmente, nenhum comando é executado.

Na segunda forma, o conjunto de comandos é executado ao menos uma vez. O ciclo de repetições dos comandos só será interrompido quando a expressão booleana passar a ser verdadeira.

O comando *for* já é muito conhecido e nos limitaremos a dizer que é possível substituir o termo to por downto se desejarmos que a variável de controle decresça ao invés de aumentar durante o ciclo.

Comandos condicionais

Os comandos condicionais existentes são:

if <expressão boleana> then <comando>

if <expressão booleana> then <comando> else <comando>

```
case <expressão> of
   <lista de possíveis valores da expressão> : <comando>;
                          :
   <lista de possíveis valores da expressão> : <comando>;
end
```

O comando *if* é mais um daqueles que dispensam explicações.

O comando *case*, por sua vez, dependendo do valor da expressão do cabeçalho, executa um determinado comando diferente.

Suponhamos, por exemplo, que uma variável *i*, inteira, pode assumir, num determinado ponto de um programa, os valores 1, 2, 3, 4, 5 e 6, assim, o comando abaixo é válido:

```
case i of
   1 .. 3 : write('i >= 3');
   4      : write('i = 4');
   5,6    : write('i >= 5');
end
```

APÊNDICE B

DOCUMENTAÇÃO DOS PROGRAMAS APRESENTADOS

SEÇÃO 1 PRINCIPAIS CONSTANTES E VARIÁVEIS UTILIZADAS

Fornecemos abaixo uma lista das principais constantes e variáveis empregadas pelos programas apresentados ao longo do texto.

Os índices que seguem alguns termos significam que eles só são encontrados num ou noutro programa (numa referência ao capítulo em que este foi descrito). As variáveis comuns não possuem índice.

Constantes

maxit = 10000	número máximo de iterações de cada fase do simplex
novox = 10	número de iterações entre cálculos sucessivos de x
mu = 10	limitante superior dos elementos de U oriundos das atualizações

Variáveis Inteiras

nbloc	número de blocos da matriz A;
nlest	número de linhas de estoque da matriz A;
tcol	número total de colunas da base B;
bcol	número de colunas dos blocos de B;
tlin	número total de linhas dos blocos de A;
$nfart^3$	número total de colunas de folga e artificiais;
ufim	última posição ocupada no vetor que guarda U;
ufimf	idem, mas mas com respeito apenas à fatoração;
at	número de atualizações desde a última fatoração;
maxat	número máximo de atualizações entre fatorações;
niter	número de iterações do simplex já executadas;
ent	coluna de A que deverá entrar na base;

sai	coluna de B que irá sair da base;
AtOt[9]	iteração na qual se obteve o melhor fator de economia de tempo;
MemMin[9]	Quantidade mínima de memória necessária para guardar os acréscimos a L;
TamAcU[9]	Tamanho do vetor utilizado para armazenar U;
UlBloE[5]	Bloco da última variável a entrar na base
UlVarE[5]	Índice da última variável a entrar na base

Variáveis reais

norma	norma do erro no cálculo de *b*;
z	valor da função objetivo;
ctobj	constante de escalamento da função objetivo;
FatEcon[9]	fator de economia de tempo;

Variáveis booleanas

fase1	indica se estamos ou não na fase 1 do simplex;
nfat	indica se é necessária uma nova fatoração da base;
JaHaX[5]	indica se o vetor intermediário x do sistema $B^T z=y$ está disponível

Vetores inteiros com índices entre 0 e nbloc+1

gcol[i][3] índice da última coluna do bloco *i* de A;

ncan[i][3] índice da primeira coluna correspondente a uma variável de folga de uma restrição canalizada do bloco *i* de A;

nfol[i][3] índice da primeira coluna do bloco *i* correspondente a uma variável de folga ou excesso (não pertencente a uma restrição canalizada);

nart[i][3] índice da primeira coluna correspondente a uma variável artificial do bloco *i* de A;

cest[i][3] soma do número de colunas correspondentes às variáveis "originais" (isto é, excluindo as variáveis de folga, excesso e artificiais) dos blocos 1 até *i* de A;

ncol[i][3] índice da última coluna do bloco *i* da base;

nlin[i][3] índice da última linha do bloco *i* da base e da matriz A;

IndBlo[i][5] índice do bloco original do problema que, no instante, ocupa a *i*-ésima posição na matriz A

Vetores e variáveis especiais

l^[k]	*k*-ésimo elemento não nulo da matriz L (triangular inferior) oriunda da fatoração, armazenada seqüencialmente por colunas;
u^[k]	valor do único elemento não nulo acima da diagonal da *j*-ésima matriz triangular superior unitária U_j;
lac[j]$^{\wedge 3}$	apontador do início da lista ligada que armazena os elementos da coluna *j* de L situados abaixo do bloco junto à diagonal que contém esta coluna;
lacf[j]$^{\wedge 3}$	apontador que guarda a posição de *lac[j]*$^{\wedge}$ logo após a fatoração (antes das atualizações);
pnil$^{\wedge 3}$	apontador que indica o final de todas as listas ligadas *lac*;
upos[k].x	linha do elemento não nulo acima da diagonal de cada matriz triangular unitária U originária das atualizações;
upos[k].y	semelhante a *upos[k].x*, fornece a coluna do mesmo elemento;
upos[k].p	Indica se foi necessária uma permutação de colunas antes que fosse aplicada a matriz U_k;
lpos[j]	posição, no vetor *l*, do primeiro elemento não nulo (ou seja do elemento da diagonal) da coluna *j* da matriz L;
cpos[j]	indica a posição da coluna *j* da base após as permutações da fatoração;

icnl[j] semelhante a *cpos*, indica a posição das colunas que sobram nos blocos não quadrados da base;

cabas[j] indica qual coluna de A ocupa a *j*-ésima posição dentre as colunas da base;

a[i][3] *i*-ésimo bloco da matriz A, armazenado sequencialmente por linhas e considerando apenas as colunas "originais" do problema;

aest[i][3] i-ésima linha de A correspondente a uma restrição de estoque;

afart[k][3] indica qual a linha correspondente à *k*-ésima variável de folga, excesso ou artificial do problema (*abs(afart[k])*) e qual o tipo desta variável. Se *afart[k]*<0 então a variável é excesso

luest[j][k] matriz que contém os elementos de L e U correspondentes às últimas *nlest* linhas e colunas;

esc[i][5] indica o fator de escalamento da *i*-ésima linha de acoplamento do problema

matriz[i][5] registro que armazena todos os dados importantes relativos ao *i*-ésimo bloco do problema.

——.*ncor*[5] número de colunas "originais" do bloco;

——.*ncca*[5] número de rtestrições canalizadas;

——.*ncfo*[5] número de variáveis de folga de restrições não canalizadas;

——.*ncar*[5] número de variáveis artificiais do bloco;

——.*ncol*[5] número de colunas do bloco que pertencem à base;

——.*nlin*[5] número de linhas do bloco;

——.*prod*[5] valor dos elementos não nulos do acoplamento;

——.*est*[6] índices das colunas que têm elementos no estoque;

——.x^5 semelhante ao vetor *vu* descrito abaixo;

——.sca^5 fator de escalamento das linhas do bloco;

——.dad^5 semelhante ao vetor *a*, já citado;

——.$lest^5$ armazena os elementos das linhas de acoplamento da matriz L

Vetores booleanos

$fobas[j]^3$ indica se a coluna *j* de A está fora da base;

$lim[j]^3$ indica se a *j*-ésima variável (x) do problema atingiu o seu limite superior;

Vetores reais

$linf[j]^3$ limite inferior da variável *j* do problema;

$lsup[j]^3$ diferença entre os limites superior e inferior da mesma variável;

$obj1[j]^3$ *j*-ésimo coeficiente da função objetivo que está sendo usada no simplex;

$obj2[j]^3$ *j*-ésima componente do vetor *c*, função objetivo original do problema;

$vu[i]^3$	i-ésima componente básica do vetor solução x do problema;
$bmin[i]^3$	Limite inferior da i-ésima restrição do problema;
$b[i]^3$	diferença entre os limites superior e inferior da i-ésima restrição do problema;

PROCEDURE ATUALIZACAO

ENTRADA — *ent*, índice da coluna que entra na base, *sai*, índice da coluna que sai e *blent* e *blsai*, seus respectivos blocos.

CONTEÚDO — Substitui uma coluna da base por outra de A atualizando as matrizes L e U oriundas da fatoração.

SAÍDA — As matrizes L, U e P atualizadas.

PROCEDURE BTZY

ENTRADA — Vetor y.

CONTEÚDO — Resolve o sistema $B^T z = y$.

SAÍDA — Vetor z.

SUBROTINAS REQUISITADAS — *UtXY* e *LtZX*.

PROCEDURE BZY

ENTRADA — Vetor y.

CONTEÚDO — Resolve o sistema $B.z = y$.

SAÍDA — Vetor z.

SUBROTINAS REQUISITADAS — *LXY* e *UZX*.

FUNCTION CALCMEM[a]

CONTEÚDO Verifica se há memória disponível para uma atualização.

SAÍDA Se não houver memória a função retorna *true*, caso contrário retorna *false*.

FUNCTION CALCNORMA

CONTEÚDO Calcula $\|b - Ax\|_\infty$.

SAÍDA Se a norma for menor que uma tolerância, então retorna *true*, caso contrário retorna *false*.

FUNCTION CALCTEMPO[a]

CONTEÚDO Verifica se a próxima atualização se justifica sob o ponto de vista do consumo de tempo.

SAÍDA Se for conveniente refatorar a base a função retorna *true*, caso contrário retorna *false*.

PROCEDURE CALC_X

CONTEÚDO Calcula o valor das componentes da solução no vértice atual.

SAÍDA O vetor x e o valor de z.

SUBROTINAS REQUISITADAS *BZY* e *Calc_Z*.

PROCEDURE CALC_Z

CONTEÚDO Calcula $z = cx$.

SAÍDA O valor de z.

PROCEDURE CICLO

CONTEÚDO — Determina a variável que entra e a que sai da base. Atualiza o valor de x. Atualiza a base. Termina o programa ser tiver sido excedido o limite de iterações.

SAÍDA — Fornece *ent*, índice da coluna que entra, e, *sai*, índice da coluna que sai da base, para que a rotina principal possa detectar se há ou não sulução para o problema, se a solução é ilimitada ou se há um vértice ótimo.

SUBROTINAS REQUISITADAS — São chamadas as rotinas *Atualização*, *BtZY*, *BZY*, *CalcNorma*, *CalcTempo*[3], *CalcMem*[3], *NovaFatoração*, *QuemEntra* e *QuemSai*

PROCEDURE ESCALAMENTO

CONTEÚDO — Escala a matriz tecnológica A.

SAÍDA — A matria A escalada por linhas.

PROCEDURE FATORAÇAO

CONTEÚDO Decompõe a matriz base B em submatrizes L, triangular inferior, e U, trialgular superior.

SAÍDA As matrizes L, U e P.

PROCEDURE LEDADOS

CONTEÚDO Lê os dados do problema armazenados em disco.

SAÍDA A matriz A, os vetores b, c e x^{sup}.

PROCEDURE LTZX

ENTRADA Vetor X.

CONTEÚDO Resolve $L^T z = x$.

SAÍDA Vetor z.

PROCEDURE LXY

ENTRADA Vetor y.

CONTEÚDO Resolve o sistema $L.x = y$.

SAÍDA Vetor x.

PROCEDURE MUDAORDEM[5]

ENTRADA O bloco i que passará à última posição da matriz

CONTEÚDO Permuta as posições correspondentes aos blocos i e *nbloc* no vetor *IndBlo* e corrige os valores dos elementos do vetor *cabas*.

SAÍDA Os vetores *cabas* e *IndBlo* atualizados.

PROCEDURE NOVAFATORAÇÃO

ENTRADA — É preciso fornecer o índice da coluna que entra e o daquela que sai da base.

CONTEÚDO — Reordena as colunas da base. Substitui a coluna da variável que sai da base pela da que entra nas matrizes triangulares L e U através de uma nova fatoração. Calcula o vetor solução x.

SAÍDA — A nova base, fatorada.

SUBROTINAS REQUISITADAS — *OrdenaCabas*, *Fatoração*, *Calc_X* e *MudaOrdem*[5].

PROCEDURE OBJETIVO1

CONTEÚDO — Cria a função objetivo da fase 1 do simplex. Determina a base inicial do algoritmo. Atribui valores iniciais aos elementos de x.

SAÍDA — A função objetivo da fase 1 do simplex.

PROCEDURE OBJETIVO2

CONTEÚDO — Substitui a função objetivo da fase 1 pelo vetor c original do problema. Elimina as variáveis artificiais. Recalcula z.

SAÍDA — A função objetivo da fase 2 do simplex.

SUBROTINAS REQUISITADAS — *Calc_Z*.

PROCEDURE ORDENACABAS

CONTEÚDO — Ordena as colunas da base de acordo com seus índices (ordem crescente).

SAÍDA — As colunas da base ordenadas por índice.

PROCEDURE QUEMENTRA

CONTEÚDO Determina a coluna que deverá entrar na base na iteração atual do simplex.

SAÍDA *ent*, índice da coluna que entra na base, e *blent*, seu bloco. Se *ent* = 0, então a solução é ótima.

PROCEDURE QUEMSAI

CONTEÚDO Determina a coluna que irá sair da base nesta iteração do simplex. Atualiza o vetor x.

SAÍDA *sai*, índice da coluna que sai da base e *blsai*, seu bloco. Se *sai* = 0 então a solução do problema é ilimitada.

PROCEDURE UTXY

ENTRADA O vetor y.

CONTEÚDO Resolve o sistema $U^T x = y$.

SAÍDA O vetor x.

PROCEDURE UZX

ENTRADA Vetor X.

CONTEÚDO Resolve o sistema $U.z = x$.

SAÍDA Vetor z.

Os programas apresentados podem ser executados diretamente a partir das listagens que serão fornecidas no próximo apêndice.

Mas para que eles funcionem corretamente será preciso reservar memória e atribuir valores iniciais a alguns vetores definidos recentemente. Fornecemos abaixo uma lista destes vetores, separados por algoritmo.

Programa do capítulo III

Neste programa será preciso alocar espaço e fornecer valores iniciais às variáveis: *nbloc*, *nlest*, *tcol*, *bcol*, *tlin*, *nfart*, *gcol*, *ncan*, *nfol*, *nart*, *cest*, *nlin*, *a*, *aest*, *afart*, *lsup*, *linf*, *objz*, *b*, *bmin* e *esc*.

Os vetores para os quais basta reservar memória são: *l*, *u*, *lacf*, *upos*, *lpos*, *cpos*, *cabas*, *objx*, *pi*, *vu*, *g*, *fobas* e *lim*.

Programa do capítulo V

Neste programa também será preciso atribuir valores iniciais às variáveis: *nbloc*, *nlest*, *tcol*, *bcol* e *tlin*. Da mesma forma, dentro do vetor matriz, para cada bloco é necessário definir: *nlin*, *ncor*, *ncca*, *ncfo*, *ncar*, *prod*, *est*, *afart*, *linf*, *lsup*, *obj2*, *b*, *bmin* e *dad* (alocando memória quando a variável for um apontador de um vetor).

Os vetores que precisam apenas ter espaço reservado são: *l*, *u*, *upos*, *lpos*, *cpos*, *cabas*, *pi*, *g* e *xtmp*.

APÊNDICE C

LISTAGENS

Aos leitores interessados em implantar os programas comentados neste texto, ou àqueles que desejam estudar os algoritmos mais detalhadamente, fornecemos abaixo as listagens deles, das quais excluímos apenas a parte correspondente à leitura de dados, que, ao nosso ver, deve ser elaborada segundo critérios definidos pelo usuário.

```
Program Simplex_do_capitulo_3(input,output);


Const
  maxblo  =   100;
  maxtmn  =   102;
  maxeba  =  2500;
  maxinq  =  6000;
  maxces  =  3600;
  maxcot  =  3000;
  maxest  =    10;
  maxfol  =  3000;
  maxelU  = 16000;
  maxelL  = 16000;
  maxeaU  = 10000;
  maxlac  =    50;
  inf     =  1e20;
  zeroa   = 1e-10;
  zerox   =  1e-5;
  zeroz   =  1e-5;
  zerob   = 1e-10;
  zerol   = 1e-10;
  zeroesc = 1e-10;
  zeroent =  1e-5;
  zerosai =  6e-5;
  zerosis = 1e-10;
  zerou   = 1e-10;
  zerolu  =  1e-8;
  zeropi  =  1e-7;
  zeroac. =  1e-7;
  zeronor =  6e-5;
  maxit   = 10000;
  novox   =    10;
  mu      =  10.0;
  tamreal =     4;
```

```
Type
  xyp   = record
            x,y,p: integer;
          end;
  tymn  = array[0..maxtmn] of integer;
  tyL   = array[1..maxelL] of single;
  tyU   = array[1..maxelU] of single;
  typU  = array[1..maxeaU] of xyp;
  typ   = array[0..maxcot] of integer;
  tyAp  = array[1..maxblo] of ^tyA;
  tyA   = array[1..maxeba] of single;
  tyfol = array[1..maxfol] of integer;
  tyobj = array[1..maxing] of single;
  tyvet = array[0..maxcot] of single;
  tyest = array[1..maxces] of single;
  tyesp = array[1..maxest] of ^tyest;
  tyLUe = array[1..maxest,1..maxest] of single;
  tyicn = array[1..maxest] of integer;
  tyfob = array[1..maxing] of boolean;
  tydad = array[1..maxlac] of single;
  typac = ^tyac;
  tyac  = record
            blo: integer;
            pro: typac;
            dad: tydad;
          end;
  tyLac = array[1..maxcot] of typac;

Var
  nbloc,nlest,tcol,bcol,tlin,nfart,at,
  ufim,ufimf,maxat,niter,ent,sai,AtOt                                  :  integer;
  MemMin,TamAcU                                                        :  word;
  MemDsp                                                               :  LongInt;
  TmpFat,TmpAt,TmpAtOt,TmpAtAc,TmpSisOr,TmpBZY,TmpNorma,FatEcon:  single;
  norma,ctobj,zinic,z                                                  :  double;
  gcol,ncan,nfol,nart,cest,ncol,nlin                                   :  tymn;
  l                                                                    : ^TyL;
  u                                                                    : ^TyU;
  lac,lacf                                                             :  tyLac;
  pmark                                                                :  typac;
  upos                                                                 : ^typU;
  lpos,cpos,cabas                                                      : ^typ;
  a                                                                    :  tyAp;
  icn                                                                  :  tyicn;
  luest                                                                :  tyLUe;
  pnil                                                                 :  typac;
  aest                                                                 :  tyesp;
  afart                                                                : ^tyfol;
  fasel,nfat,vermem                                                    :  boolean;
  fobas,lim                                                            : ^tyfob;
```

```
  lsup,linf,obj1,obj2                                    : ^tyobj;
  pi,vu,b,bmin,g,esc                                     : ^tyvet;

Procedure LeDados;

  var i: word;

  begin
    {esta subrotina contem a leitura dos dados do problema e o}
    {dimensionamento dos vetores do programa, terminando com: }
    pi^[0]:=1;
    vu^[0]:=inf;
    sai:=31000;
    cabas^[tcol+1]:=31000;
    cabas^[0]:=0;
    nfart:=gcol[nbloc+1]-cest[nbloc];
    GetMem(pnil,6);
    pnil^.blo:=nbloc+2;
    pnil^.pro:=pnil;
    for i:=1 to tlin+1 do
      begin
        GetMem(lacf[i],6+word(nlest)*tamreal);
        lacf[i]^.blo:=nbloc+1;
        lacf[i]^.pro:=pnil;
        lac[i]:=lacf[i];
      end;
  end; {LeDados}

Procedure Escalamento;

  var
    k,l,ncb,i1: integer;
    t         : single;

  Procedure EscBlocos;

    var bloco,i,j: integer;

    begin
      for bloco:=1 to nbloc do
        begin
          k:=ncan[bloco];
          i1:=cest[bloco];
          ncb:=i1-cest[bloco-1];
          for i:=nlin[bloco-1]+1 to nlin[bloco] do
            begin
              t:=zeroesc;
              l:=(i-nlin[bloco-1]-1)*ncb;
              for j:=l+1 to l+ncb do
                if abs(a[bloco]^[j])>t then t:=abs(a[bloco]^[j]);
```

```
            for j:=l+1 to l+ncb do
              if abs(a[bloco]^[j]/t)<zeroa
                then a[bloco]^[j]:=0.0
                else a[bloco]^[j]:=a[bloco]^[j]/t;
            b^[i]:=b^[i]/t;
            esc^[i]:=1.0/t;
            while (k<(nfol[bloco]-1)) and
                  (abs(afart^[k-il])<(i-nlin[bloco-1])) do inc(k);
            if (abs(afart^[k-il])=(i-nlin[bloco-1])) and (k<nfol[bloco])
              then
                begin
                  linf^[k]:=linf^[k]/t;
                  lsup^[k]:=lsup^[k]/t;
                end;
          end;
      end;
  end; {EscBlocos}

Procedure EscEstoque;

  var i,j: integer;

  begin
    k:=ncan[nbloc+1];
    il:=cest[nbloc+1];
    for i:=1 to nlest do
      begin
        t:=zeroesc;
        for j:=1 to cest[nbloc] do
          if abs(aest[i]^[j])>t then t:=abs(aest[i]^[j]);
        b^[tlin+i]:=b^[tlin+i]/t;
        esc^[tlin+i]:=1.0/t;
        for j:=1 to cest[nbloc] do
          if abs(aest[i]^[j]/t)<zeroa
            then aest[i]^[j]:=0.0
            else aest[i]^[j]:=aest[i]^[j]/t;
        while (k<(nfol[nbloc+1]-1)) and (abs(afart^[k-il])<i) do inc(k);
        if (abs(afart^[k-il])=i) and (k<nfol[nbloc+1]) then
          begin
            linf^[k]:=linf^[k]/t;
            lsup^[k]:=lsup^[k]/t;
          end;
      end;
  end; {EscEstoque}
```

```
  Procedure EscObjetivo;

    var i: integer;

    begin
      ctobj:=zeroesc;
      for i:=1 to cest[nbloc+1] do
        if abs(obj2^[i])>ctobj then ctobj:=abs(obj2^[i]);
      for i:=1 to cest[nbloc+1] do
        if abs(obj2^[i]/ctobj)<zeroa
          then obj2^[i]:=0.0
          else obj2^[i]:=obj2^[i]/ctobj;
    end; {EscObjetivo}

  begin
    EscBlocos;
    EscEstoque;
    EscObjetivo;
  end; {Escalamento}

Procedure Fatoracao;

  var
    pivo,linbl,uinbl,pinbl,nlibl,ncibl,ncabl,pinbl2,
    lcont,ucont,pcont,econt,s,ss,tt,pt,cia          : integer;
    t,t1,t2                                         : single;

  Procedure FatBlocos;

    var i,j,k,kk: integer;

    begin
      lcont:=1;
      ucont:=1;
      pcont:=1;
      econt:=0;
      pinbl2:=1;
      for i:=1 to nbloc do
        begin
          uinbl:=ucont;
          pinbl:=pcont;
          linbl:=lcont;
          nlibl:=nlin[i]-nlin[i-1];
          ncibl:=ncol[i]-ncol[i-1];
          ncabl:=cest[i]-cest[i-1];
          cia:=cest[i-1];
          for j:=pinbl to pinbl+ncibl-1 do cpos^[j]:=pinbl2-pinbl+j;
          for j:=1 to nlibl do
            begin
              tt:=(j-1)*ncabl-gcol[i-1];
```

```
for k:=j to ncibl do
  begin
    ss:=cabas^[cpos^[pinbl+k-1]];
    if ss<ncan[i]
      then u^[ucont+k-j]:=a[i]^[tt+ss]
      else if abs(afart^[ss-cest[i]])=j
             then u^[ucont+k-j]:=afart^[ss-cest[i]]/j
             else u^[ucont+k-j]:=0.0;
  end;
s:=uinbl-1;
for k:=1 to j-1 do
  begin
    t:=l^[lpos^[pinbl+k-1]+j-k];
    if abs(t)>zerou then
      for kk:=j to ncibl do
        u^[ucont+kk-j]:=u^[ucont+kk-j]+u^[s+kk-k]*t;
    s:=s+ncibl-k;
  end;
pivo:=ucont;
for k:=ucont+1 to ucont+ncibl-j do
  if abs(u^[k])>abs(u^[pivo]) then pivo:=k;
if abs(u^[pivo])<zerolu then saida(BaseSingular);
s:=uinbl-1;
for k:=1 to j-1 do
  begin
    kk:=pivo-ucont+s+j-k;
    t:=u^[kk];
    u^[kk]:=u^[s+j-k];
    u^[s+j-k]:=t;
    s:=s+ncibl-k;
  end;
t:=u^[pivo];
u^[pivo]:=u^[ucont];
s:=cpos^[pcont];
cpos^[pcont]:=cpos^[pcont+pivo-ucont];
cpos^[pcont+pivo-ucont]:=s;
for k:=ucont to ucont+ncibl-j-1 do
  begin
    u^[k]:=-u^[k+1]/t;
    if abs(u^[k])<zerou then u^[k]:=0.0;
  end;
l^[lcont]:=t;
lpos^[pcont]:=lcont;
kk:=cpos^[pinbl+j-1];
ss:=cabas^[kk]-gcol[i-1];
if ss<=ncabl
  then for k:=j+1 to nlibl do
         l^[lcont+k-j]:=a[i]^[(k-1)*ncabl+ss]
  else for k:=j+1 to nlibl do l^[lcont+k-j]:=0.0;
```

```
            with lacf[pcont]^ do
              if ss<=ncabl
                then for k:=1 to nlest do dad[k]:=aest[k]^[ss+cia]
                else for k:=1 to nlest do dad[k]:=0.0;
            s:=0;
            ss:=0;
            for k:=1 to j-1 do
              begin
                t:=u^[uinbl+ss+j-k-1];
                if abs(t)>zerou then
                  begin
                    for kk:=j+1 to nlibl do
                      l^[lcont+kk-j]:=l^[lcont+kk-j]+t*l^[linbl+kk+s-1];
                    with lacf[pcont]^ do
                      for kk:=1 to nlest do
                        dad[kk]:=dad[kk]+t*lacf[pinbl+k-1]^.dad[kk];
                  end;
                s:=s+nlibl-k;
                ss:=ss+ncibl-k;
              end;
            lcont:=lcont+nlibl-j+1;
            Inc(pcont);
            ucont:=ucont+ncibl-j;
          end;
        for j:=1 to ncibl-nlibl do
          begin
            icn[econt+j]:=cpos^[pcont+j-1];
            ss:=cabas^[icn[econt+j]]-gcol[i-1];
            if ss<=ncabl
              then for k:=1 to nlest do luest[k,econt+j]:=aest[k]^[ss+cia]
              else for k:=1 to nlest do luest[k,econt+j]:=0.0;
          end;
        econt:=econt+ncibl-nlibl;
        pinbl2:=pinbl2+ncibl;
      end;
  end; {FatBlocos}

Procedure AplFatLuest;

  var i,j,k,kk: integer;

  begin
    for j:=1 to tcol-bcol do
      begin
        icn[econt+j]:=bcol+j;
        for k:=1 to nlest do luest[k,econt+j]:=0.0;
        ss:=afart^[cabas^[bcol+j]-cest[nbloc+1]];
        luest[abs(ss),econt+j]:=ss/abs(ss);
      end;
```

```
      for j:=tlin+1 to tcol do
        begin
          cpos^[j]:=icn[j-tlin];
          icn[j-tlin]:=-j+tlin;
        end;
      ufim:=ucont-1;
      ufimf:=ufim;
      uinbl:=1;
      ucont:=1;
      pinbl:=1;
      for i:=1 to nbloc do
        begin
          nlibl:=nlin[i]-nlin[i-1];
          ncibl:=ncol[i]-ncol[i-1];
          for j:=nlibl+1 to ncibl do
            begin
              s:=0;
              ss:=0;
              for k:=1 to nlibl do
                begin
                  t:=u^[uinbl+ss+j-1-k];
                  if abs(t)>zerou then
                    for kk:=1 to nlest do
                      luest[kk,ucont]:=luest[kk,ucont]
                                       +t*lacf[pinbl+k-1]^.dad[kk];
                  s:=s+nlibl-k;
                  ss:=ss+ncibl-k;
                end;
              Inc(ucont);
            end;
          pinbl:=pinbl+nlibl;
          uinbl:=uinbl+((nlibl*nlibl-nlibl) div 2)+(ncibl-nlibl)*nlibl;
        end;
    end; {AplFatLuest}

  Procedure FatEstoque;

    var j,k,kk: integer;

    begin
      for j:=1 to nlest-1 do
        begin
          pivo:=j;
          for k:=j+1 to nlest do
            if abs(luest[j,k])>abs(luest[j,pivo]) then pivo:=k;
          t1:=luest[j,pivo];
          if pivo<>j then
            begin
              tt:=cpos^[tlin+j];
              cpos^[tlin+j]:=cpos^[tlin+pivo];
```

```
            cpos^[tlin+pivo]:=tt;
            tt:=icn[j];
            icn[j]:=icn[pivo];
            icn[pivo]:=tt;
            for k:=1 to nlest do
              begin
                t2:=luest[k,pivo];
                luest[k,pivo]:=luest[k,j];
                luest[k,j]:=t2;
              end;
          end;
        for k:=j+1 to nlest do
          begin
            t2:=-luest[j,k]/t1;
            luest[j,k]:=t2;
            if abs(t2)>zerou then
              for kk:=j+1 to nlest do
                luest[kk,k]:=luest[kk,k]+t2*luest[kk,j];
          end;
      end;
  end; {FatEstoque}

Procedure OrdenaIcn;

  var i: integer;

  begin
    for i:=1 to nlest do
      if icn[i]<0 then
        begin
          s:=i;
          tt:=-icn[i];
          while tt<>i do
            begin
              ss:=tt;
              tt:=-icn[tt];
              icn[ss]:=s;
              s:=ss;
            end;
          icn[i]:=s;
        end;
  end; {OrdenaIcn}

begin.
  FatBlocos;
  AplFatLuest;
  FatEstoque;
  OrdenaIcn;
end; {Fatoracao}
```

```
Procedure BZY(var z,y: tyvet);

  var x: tyvet;

  Procedure LXY(var x,y: tyvet);

    var
      i,j,k,m: integer;
      t      : single;
      act    : typac;

    begin
      for i:=1 to tcol do x[i]:=y[i];
      for i:=1 to nbloc do
        for j:=nlin[i-1]+1 to nlin[i] do
          begin
            k:=lpos^[j];
            x[j]:=x[j]/l^[k];
            if abs(x[j])>zerosis
              then
                begin
                  for m:=1 to nlin[i]-j do
                    x[j+m]:=x[j+m]-x[j]*l^[k+m];
                  act:=lac[j];
                  repeat
                    k:=nlin[act^.blo-1];
                    with act^ do
                      for m:=k+1 to nlin[blo] do
                        x[m]:=x[m]-x[j]*dad[m-k];
                    act:=act^.pro;
                  until act=pnil;
                end
              else x[j]:=0.0;
        end;
      for i:=1 to nlest do
        begin
          t:=x[tlin+i];
          for j:=1 to i-1 do
            t:=t-x[tlin+j]*luest[i,j];
          if abs(t)>zerosis
            then x[tlin+i]:=t/luest[i,i]
            else x[tlin+i]:=0.0;
        end;
    end; {LXY}
```

```
Procedure UZX(var z,x: tyvet);

  var
    i,j,k,ucont,uest,xinbl,ip,nlibl,ncibl: integer;
    ut                                   : xyp;
    t                                    : single;

  begin
    for i:=ufim downto ufimf+1 do
      begin
        ut:=upos^[i-ufimf];
        if ut.p<0
          then
            begin
              x[ut.x]:=x[ut.x]+x[ut.y]*u^[i];
              if abs(x[ut.x])<zerosis then x[ut.x]:=0.0;
            end
          else
            begin
              x[ut.y]:=x[ut.y]+x[ut.x]*u^[i];
              if abs(x[ut.y])<zerosis then x[ut.y]:=0.0;
            end;
        ip:=abs(ut.p);
        if ut.y<>ip then
          begin
            t:=x[ip];
            x[ip]:=x[ut.y];
            x[ut.y]:=t;
          end;
      end;
    ucont:=ufimf;
    uest:=nlest-tcol+bcol;
    for i:=nlest downto 1 do
      begin
        for j:=i+1 to nlest do
          x[tlin+i]:=x[tlin+i]+x[tlin+j]*uest[i,j];
        if abs(x[tlin+i])<zerosis then x[tlin+i]:=0.0;
      end;
    for i:=nbloc downto 1 do
      begin
        nlibl:=nlin[i]-nlin[i-1];
        ncibl:=ncol[i]-ncol[i-1];
        uest:=uest-ncibl+nlibl;
        xinbl:=nlin[i-1];
        for j:=nlibl downto 1 do
          begin
            ucont:=ucont-ncibl+nlibl;
            for k:=1 to ncibl-nlibl do
              x[xinbl+j]:=x[xinbl+j]+x[tlin+icn[uest+k]]*u^[ucont+k];
            ucont:=ucont-nlibl+j;
```

```
              for k:=j+1 to nlibl do
                x[xinbl+j]:=x[xinbl+j]+x[xinbl+k]*u^[ucont+k-j];
              if abs(x[xinbl+j])<zerosis then x[xinbl+j]:=0.0;
            end;
        end;
      for i:=1 to tcol do z[cpos^[i]]:=x[i];
    end; {UZX}

  begin
    LXY(x,y);
    UZX(z,x);
  end; {BZY}

Procedure UtXY(var x,y: tyvet);

  var
    i,j,k,ip,xinbl,ucont,uest,nlibl,ncibl: integer;
    ut                                   : xyp;
    t                                    : single;

  begin
    for i:=1 to tcol do x[i]:=y[cpos^[i]];
    xinbl:=0;
    ucont:=0;
    uest:=0;
    for i:=1 to nbloc do
      begin
        nlibl:=nlin[i]-nlin[i-1];
        ncibl:=ncol[i]-ncol[i-1];
        for j:=1 to nlibl do
          begin
            t:=x[xinbl+j];
            for k:=j+1 to nlibl do
              x[xinbl+k]:=x[xinbl+k]+t*u^[ucont+k-j];
            ucont:=ucont+nlibl-j;
            for k:=1 to ncibl-nlibl do
              x[tlin+icn[uest+k]]:=x[tlin+icn[uest+k]]+t*u^[ucont+k];
            ucont:=ucont+ncibl-nlibl;
          end;
        xinbl:=nlin[i];
        uest:=uest+ncibl-nlibl;
      end;
    for i:=1 to nlest-1 do
      for j:=i+1 to nlest do
        x[tlin+j]:=x[tlin+j]+x[tlin+i]*luest[i,j];
    for i:=1 to tlin+nlest do
      if abs(x[i])<zerosis then x[i]:=0.0;
    for i:=ufimf+1 to ufim do
      begin
        ut:=upos^[i-ufimf];
```

```
        ip:=abs(ut.p);
        if ut.y<>ip then
          begin
            t:=x[ip];
            x[ip]:=x[ut.y];
            x[ut.y]:=t;
          end;
        if ut.p<0
          then
            begin
              x[ut.y]:=x[ut.y]+x[ut.x]*u^[i];
              if abs(x[ut.y])<zerosis then x[ut.y]:=0.0;
            end
          else
            begin
              x[ut.x]:=x[ut.x]+x[ut.y]*u^[i];
              if abs(x[ut.x])<zerosis then x[ut.x]:=0.0;
            end;
      end;
  end; {UtXY}

Procedure BtZY(var z,y: tyvet);

  Procedure LtZX(var z: tyvet);

    var
      i,j,k,m: integer;
      r      : single;
      act    : typac;

    begin
      for i:=nlest downto 1 do
        begin
          r:=0;
          for j:=i+1 to nlest do
            r:=r-z[tlin+j]*luest[j,i];
          z[tlin+i]:=(z[tlin+i]+r)/luest[i,i];
          if abs(z[tlin+i])<zerosis then z[tlin+i]:=0.0;
        end;
      for i:=nbloc downto 1 do
        for j:=nlin[i] downto nlin[i-1]+1 do
          begin
            r:=0;
            act:=lac[j];
            repeat
              k:=nlin[act^.blo-1];
              with act^ do
                for m:=k+1 to nlin[blo] do
                  r:=r-z[m]*dad[m-k];
              act:=act^.pro;
```

```
            until act=pnil;
            k:=lpos^[j];
            for m:=1 to nlin[i]-j do
              r:=r-z[j+m]*l^[k+m];
            z[j]:=(z[j]+r)/l^[k];
            if abs(z[j])<zerosis then z[j]:=0.0;
          end;
      end; {LtZX}

    begin
      UtXY(z,y);
      LtZX(z);
    end; {BtZY}

  Procedure AlocaPac(var pntr: typac; tamanho: word);

    begin
      GetMem(pntr,tamanho);
      MemDsp:=MemDsp-tamanho;
    end; {AlocaPac}

  Procedure DesalocaPac(var pntr: typac; tamanho:word);

    begin
      FreeMem(pntr,tamanho);
      MemDsp:=MemDsp+tamanho;
    end; {DesalocaPac}

  Procedure Atualizacao(colent,colsai,blent,blsai: integer);

    var
      cpri,cult,ip,li,bl,bli,lan,lp,s,it,ncabl,ss,si,st: integer;
      t,tt,gt                                          : single;
      act,bct,cct,dct                                  : typac;

    Procedure CalcCt;

      var i: integer;

      begin
        for i:=1 to tcol do g^[i]:=0.0;
        g^[colsai]:=1.0;
        UtXY(pi^,g^);
      end; {CalcCt}
```

```
Procedure CalcLtil;

  var i,j,k: integer;

  Procedure CalcLtilEst;

    var j: integer;

    begin
      with lac[tlin+1]^ do
        begin
          for j:=cult-tlin to nlest do
            dad[j]:=luest[j,cult-tlin];
          for j:=1 to cult-tlin-1 do dad[j]:=0.0;
        end;
      while ip>tlin do
        if abs(pi^[ip])<=zeropi
          then Dec(ip)
          else
            begin
              Inc(ufim);
              if abs(pi^[i])<abs(pi^[ip])
                then
                  begin
                    t:=-pi^[i]/pi^[ip];
                    with lac[tlin+1]^ do
                      begin
                        for j:=ip-tlin to i-tlin-1 do
                          begin
                            dad[j]:=luest[j,ip-tlin];
                            luest[j,ip-tlin]:=luest[j,ip-tlin]*t;
                          end;
                        for j:=i-tlin to nlest do
                          begin
                            tt:=luest[j,ip-tlin];
                            luest[j,ip-tlin]:=luest[j,ip-tlin]*t+dad[j];
                            dad[j]:=tt;
                          end;
                      end;
                    upos^[ufim-ufimf].p:=ip;
                    i:=ip;
                  end
                else
                  begin
                    t:=-pi^[ip]/pi^[i];
                    upos^[ufim-ufimf].p:=cult;
                    with lac[tlin+1]^ do
                      for j:=i-tlin to nlest do
                        luest[j,ip-tlin]:=luest[j,ip-tlin]+dad[j]*t;
                  end;
```

```
            u^[ufim]:=t;
            upos^[ufim-ufimf].x:=ip;
            upos^[ufim-ufimf].y:=cult;
            Dec(ip);
          end;
      bl:=nbloc;
      li:=i;
      i:=tlin+1;
    end; {CalcLtilEst}

  Procedure CalcLtilBlo;

    var j,k: integer;

    begin
      AlocaPac(dct,6);
      while bl>0 do
        begin
          for k:=ip downto nlin[bl-1]+1 do
            if MemDsp<0
              then
                begin
                  nfat:=true;
                  exit;
                end
              else
                if abs(pi^[k])>zeropi then
                  begin
                    Inc(ufim);
                    lan:=nlin[bl-1];
                    lp:=lpos^[k]-k;
                    if abs(pi^[li])<abs(pi^[k])
                      then
                        begin
                          t:=-pi^[li]/pi^[k];
                          upos^[ufim-ufimf].p:=k;
                          if bl<bli
                            then
                              begin
                                AlocaPac(act,6+tamreal*word(nlin[bl]-lan));
                                with act^ do
                                  begin
                                    pro:=lac[i];
                                    blo:=bl;
                                    for j:=k to nlin[bl] do
                                      begin
                                        dad[j-lan]:=l^[lp+j];
                                        l^[lp+j]:=l^[lp+j]*t;
                                      end;
                                    for j:=1 to k-lan-1 do dad[j]:=0;
```

```
                end;
              lac[i]:=act;
              act:=act^.pro;
            end

          else
            with lac[i]^ do
              begin
                for j:=k to li-1 do
                  begin
                    dad[j-lan]:=l^[lp+j];
                    l^[lp+j]:=l^[lp+j]*t;
                  end;
                for j:=li to nlin[bl] do
                  begin
                    tt:=l^[lp+j];
                    l^[lp+j]:=dad[j-lan]+t*tt;
                    dad[j-lan]:=tt;
                  end;
                act:=pro;
              end;
        bct:=act;
        lac[i]^.pro:=lac[k];
        act:=lac[k];
        lac[k]:=bct;
        li:=k;
        bli:=bl;
      end
    else
      begin
        t:=-pi^[k]/pi^[li];
        upos^[ufim-ufimf].p:=cult;
        if bl=bli
          then
            begin
              with lac[i]^ do
                for j:=li to nlin[bl] do
                  l^[lp+j]:=l^[lp+j]+t*dad[j-lan];
              act:=lac[i]^.pro;
            end
          else act:=lac[i];
      end;
  u^[ufim]:=t;
  upos^[ufim-ufimf].x:=k;
  upos^[ufim-ufimf].y:=cult;
  dct^.pro:=lac[k];
  bct:=dct;
  repeat
    if act^.blo<bct^.pro^.blo
      then
```

```
                        begin
                          AlocaPac(cct,6+tamreal*
                            word(nlin[act^.blo]-nlin[act^.blo-1]));
                          with cct^ do
                            begin
                              blo:=act^.blo;
                              pro:=bct^.pro;
                              for j:=1 to nlin[blo]-nlin[blo-1] do
                                dad[j]:=act^.dad[j]*t;
                            end;
                          bct^.pro:=cct;
                          bct:=cct;
                          act:=act^.pro;
                        end
                      else if act^.blo>bct^.pro^.blo then bct:=bct^.pro;
                    if act^.blo=bct^.pro^.blo then
                      begin
                        with bct^.pro^ do
                          for j:=1 to nlin[blo]-nlin[blo-1] do
                            dad[j]:=dad[j]+act^.dad[j]*t;
                        bct:=bct^.pro;
                        act:=act^.pro;
                      end;
                  until act=pnil;
                  lac[k]:=dct^.pro;
                end;
          Dec(bl);
          ip:=nlin[bl];
        end;
      DesalocaPac(dct,6);
    end; {CalcLtilBlo}

  begin
    i:=tcol;
    while abs(pi^[i])<=zeropi do Dec(i);
    cult:=i;
    ip:=i-1;
    bli:=nbloc+1;
    if (TamAcU-ufim)<i
      then nfat:=true
      else
        begin
          if cult<=tlin
            then
              begin
                while i<=nlin[bli-1] do Dec(bli);
                if ip<=nlin[bli-1] then bl:=bli-1 else bl:=bli;
                AlocaPac(act,6+tamreal*word(nlin[bli]-nlin[bli-1]));
```

```
                with act^ do
                  begin
                    blo:=bli;
                    pro:=lac[i];
                    lar:=nlin[bli-1];
                    k:=lpos^[i];
                    for j:=i to i-lar-1 do dad[j]:=0;
                    for j:=i to nlin[bli] do
                      dad[j-lar]:=l^[k+j-i];
                  end;
                lac[i]:=act;
                li:=i;
              end
            else CalcLtilEst;
          CalcLtilBlo;
          gt:=pi^[li];
          cpri:=li;
          act:=lac[i];
        end;
    end; {CalcLtil}

  Procedure FormaLsust;

    var i,j: integer;

    Procedure LeMatrizA(var vet: tydad);

      var i: integer;

      begin
        if st<=ncabl
          then
            for i:=1 to nlin[ss]-nlin[ss-1] do
              vet[i]:=gt*si*a[ss]^[(i-1)*ncabl+st]
          else
            begin
              for i:=1 to nlin[ss]-nlin[ss-1] do vet[i]:=0;
              i:=afart^[st+gcol[ss-1]-cest[ss]];
              vet[abs(i)]:=gt*si*i/abs(i);
            end;
      end; {LeMatrizA}

    begin
      for j:=1 to 2 do
       begin
          if j=1
            then
              begin
                ss:=blent;
                st:=colent-gcol[blent-1];
```

```
            si:=1;
          end
        else
          begin
            ss:=blsai;
            st:=cabas^[colsai]-gcol[blsai-1];
            si:=-1;
          end;
      if ss<=nbloc
        then
          begin
            ncabl:=cest[ss]-cest[ss-1];
            if ss<act^.blo
              then
                begin
                  cpri:=nlin[ss-1]+1;
                  AlocaPac(bct,6+tamreal*word(nlin[ss]-cpri+1));
                  bct^.blo:=ss;
                  bct^.pro:=act;
                  LeMatrizA(bct^.dad);
                  act:=bct;
                end
              else
                begin
                  if cpri>nlin[ss-1]+1 then cpri:=nlin[ss-1]+1;
                  bct:=act;
                  while bct^.pro^.blo<=ss do bct:=bct^.pro;
                  if bct^.blo=ss
                    then with bct^ do
                      if st<=ncabl
                        then
                          for i:=1 to nlin[ss]-nlin[ss-1] do
                            dad[i]:=dad[i]+gt*si*a[ss]^[(i-1)*ncabl+st]
                        else
                          begin
                            i:=afart^[st+gcol[ss-1]-cest[ss]];
                            dad[abs(i)]:=dad[abs(i)]+gt*si*i/abs(i);
                          end
                    else
                      begin
                       AlocaPac(cct,6+tamreal*word(nlin[ss]-nlin[ss-1]));
                       cct^.blo:=ss;
                       cct^.pro:=bct^.pro;
                       LeMatrizA(cct^.dad);
                       bct^.pro:=cct;
                      end;
                end;
          end
        else bct:=act;
    end;
```

```
    while bct^.blo<nbloc+1 do bct:=bct^.pro;
    with bct^ do
      for j:=2 downto 1 do
        begin
          if j=1 then
            begin
              ss:=blent;
              si:=1;
              st:=colent-gcol[blent-1];
            end;
          lan:=cest[ss-1];
          if st<=(cest[ss]-lan)
            then
              for i:=1 to nlest do
                dad[i]:=dad[i]+gt*si*aest[i]^[st+lan]
            else if ss>nbloc then
              begin
                i:=afart^[st+gcol[nbloc]-cest[nbloc+1]];
                dad[abs(i)]:=dad[abs(i)]+gt*si*i/abs(i);
              end;
        end;
  end; {FormaLsust}

Procedure CalcLchapeu;

  var i: integer;

  Procedure CalcLchapBlo;

    var i,j: integer;

    begin
      if cult>tlin
        then it:=tlin
        else it:=cult-1;
      i:=cpri;
      while i<=it do
        if MemDsp<0
          then
            begin
              nfat:=true;
              exit;
            end
          else
            begin
              act:=cct;
              if abs(act^.dad[i-lan])>zeroac then
                begin
                  Inc(ufim);
                  s:=lpos^[i];
```

```
if abs(l^[s])*mu>abs(act^.dad[i-lan])
  then
    begin
      t:=-act^.dad[i-lan]/l^[s];
      upos^[ufim-ufimf].p:=-cult;
      with act^ do
        for j:=i+1 to nlin[blo] do
          dad[j-lan]:=dad[j-lan]+l^[s+j-i]*t;
      bct:=lac[i];
    end
  else
    begin
      t:=-l^[s]/act^.dad[i-lan];
      upos^[ufim-ufimf].p:=-i;
      with act^ do
        for j:=i to nlin[blo] do
          begin
            tt:=dad[j-lan];
            dad[j-lan]:=l^[s+j-i]+tt*t;
            l^[s+j-i]:=tt;
          end;
      bct:=act^.pro;
      act^.pro:=lac[i];
      lac[i]:=bct;
    end;
u^[ufim]:=t;
upos^[ufim-ufimf].x:=i;
upos^[ufim-ufimf].y:=cult;
repeat
  if bct^.blo<act^.pro^.blo
    then
      begin
        AlocaPac(dct,6+tamreal*
          word(nlin[bct^.blo]-nlin[bct^.blo-1]));
        dct^.blo:=bct^.blo;
        dct^.pro:=act^.pro;
        act^.pro:=dct;
        act:=dct;
        with act^ do
          for j:=1 to nlin[blo]-nlin[blo-1] do
            dad[j]:=bct^.dad[j]*t;
        bct:=bct^.pro;
      end
    else if bct^.blo>act^.pro^.blo then act:=act^.pro;
  if act^.pro^.blo=bct^.blo then
    begin
      act:=act^.pro;
      with act^ do
        for j:=1 to nlin[blo]-nlin[blo-1] do
          dad[j]:=dad[j]+bct^.dad[j]*t;
```

```
                    bct:=bct^.pro;
                  end;
              until bct=pnil;
            end;
          if i=nlin[cct^.blo]
            then
              begin
                bct:=cct;
                cct:=cct^.pro;
                DesalocaPac(bct,6+tamreal*
                            word(nlin[bct^.blo]-nlin[bct^.blo-1]));
                lan:=nlin[cct^.blo-1];
                i:=lan+1;
              end
            else Inc(i);
        end;
  end; {CalcLchapBlo}

Procedure CalcLchapEst;

  var i,j: integer;

  begin
    if cpri>tlin
      then it:=cpri-tlin
      else it:=1;
    for i:=it to cult-tlin-1 do
      with cct^ do
        if abs(dad[i])>zeroac then
          begin
            Inc(ufim);
            if abs(luest[i,i])*mu>abs(dad[i])
              then
                begin
                  t:=-dad[i]/luest[i,i];
                  upos^[ufim-ufimf].p:=-cult;
                  for j:=i+1 to nlest do
                    dad[j]:=dad[j]+luest[j,i]*t;
                end
              else
                begin
                  t:=-luest[i,i]/dad[i];
                  upos^[ufim-ufimf].p:=-i-tlin;
                  for j:=i to nlest do
                    begin
                      tt:=dad[j];
                      dad[j]:=luest[j,i]+tt*t;
                      luest[j,i]:=tt;
                    end;
                end;
```

```
                u^[ufim]:=t;
                upos^[ufim-ufimf].x:=i+tlin;
                upos^[ufim-ufimf].y:=cult;
              end;
        for i:=cult-tlin to nlest do
          luest[i,cult-tlin]:=cct^.dad[i];
        lac[tlin+1]:=cct;
      end; {CalcLchapEst}

    begin
      lan:=nlin[act^.blo-1];
      cct:=act;
      if (TamAcU-ufim)<(cult-cpri)
        then nfat:=true
        else
          begin
            if cpri<=tlin then CalcLchapBlo;
            if not nfat then
              begin
                if cult>tlin
                  then CalcLchapEst
                  else
                    begin
                      s:=lpos^[cult];
                      lan:=nlin[cct^.blo-1];
                      with cct^ do
                        for i:=cult to nlin[blo] do
                          l^[s+i-cult]:=dad[i-lan];
                      lac[cult]:=cct^.pro;
                      DesalocaPac(cct,6+tamreal*word(nlin[cct^.blo]-lan));
                    end;
              end;
          end;
    end; {CalcLchapeu}

  begin
    TmpAt:=TempoSis;
    Inc(at);
    CalcCt;
    CalcLtil;
    if not nfat then
      begin
        FormaLsust;
        cabas^[colsai]:=colent;
        CalcLchapeu;
      end;
    TmpAt:=TempoSis-TmpAt;
  end; {Atualizacao}
```

```
Procedure CalcZ;

  var i,j: integer;

  begin
    z:=zinic;
    for i:=1 to gcol[nbloc+1] do
      if lim^[i] then z:=z+lsup^[i]*obj1^[i];
    for i:=1 to tcol do
      z:=z+vu^[i]*obj1^[cabas^[i]];
  end; {CalcZ}

Procedure CalcX;

  var
    i,j,tp1,tp2,tp3,t: integer;
    tt               : single;

  begin
    for i:=1 to tcol do g^[i]:=b^[i];
    for i:=1 to nbloc+1 do
      begin
        tp1:=nlin[i-1]+1;
        tp3:=gcol[i-1];
        for j:=tp3+1 to ncan[i]-1 do
          if lim^[j] then
            begin
              tt:=lsup^[j];
              tp2:=cest[i]-cest[i-1];
              for t:=tp1 to nlin[i] do
                g^[t]:=g^[t]-tt*a[i]^[(t-tp1)*tp2+j-tp3];
              tp2:=cest[i-1]-tp3;
              for t:=tlin+1 to tcol do
                g^[t]:=g^[t]-tt*aest[t-tlin]^[j+tp2];
            end;
        for j:=ncan[i] to nfol[i]-1 do
          if lim^[j] then
            begin
              t:=afart^[j-cest[i]];
              g^[nlin[i-1]+abs(t)]:=g^[nlin[i-1]+abs(t)]-lsup^[j]*t/abs(t);
            end;
      end;
    TmpBZY:=TempoSis;
    BZY(vu^,g^);
    TmpBZY:=TempoSis-TmpBZY;
    CalcZ;
  end; {CalcX}
```

```
Procedure Objetivo1;

  var i,j,k,tp2: integer;

  begin
    fase1:=true;
    niter:=0;
    maxat:=59;
    zinic:=0;
    for i:=1 to gcol[nbloc+1] do obj1^[i]:=0.0;
    k:=0;
    for i:=1 to nbloc+1 do
      begin
        tp2:=cest[i];
        for j:=gcol[i-1]+1 to nfol[i]-1 do fobas^[j]:=true;
        for j:=nfol[i] to nart[i]-1 do
          if afart^[j-tp2]>0
            then
              begin
                inc(k);
                cabas^[k]:=j;
                fobas^[j]:=false;
              end
            else fobas^[j]:=true;
        for j:=nart[i] to gcol[i] do
          begin
            obj1^[j]:=1;
            fobas^[j]:=false;
            inc(k);
            cabas^[k]:=j;
          end;
      end;
    for i:=1 to gcol[nbloc+1] do lim^[i]:=false;
  end; {Objetivo1}

Procedure Objetivo2;

  var i,j,tp: integer;

  begin
    niter:=0;
    fase1:=false;
    sai:=31000;
    for i:=1 to nbloc+1 do
      begin
        for j:=nart[i] to gcol[i] do
          begin;
            fobas^[j]:=false;
            obj1^[j]:=0;
          end;
```

```
        tp:=cest[i-1]-gcol[i-1];
        for j:=gcol[i-1]+1 to ncan[i]-1 do
          begin
            obj1^[j]:=obj2^[j+tp];
            zinic:=zinic+obj1^[j]*linf^[j];
          end;
      end;
    CalcZ;
  end; {Objetivo2}

Procedure OrdenaCabas; {InsertionSort}

  var i,j,v: integer;

  begin
    for i:=2 to tcol do
      begin
        v:=cabas^[i];
        j:=i;
        while cabas^[j-1]>v do
          begin
            cabas^[j]:=cabas^[j-1];
            Dec(j);
          end;
        cabas^[j]:=v;
      end;
  end; {OrdenaCabas}

Procedure NovaFatoracao(ent,sai: integer);

  var
    i,j    : integer;
    act,bct: typac;

  begin
    cabas^[sai]:=ent;
    OrdenaCabas;
    j:=0;
    for i:=1 to nbloc do
      begin
        while (cabas^[j+1]<=gcol[i]) do Inc(j);
        ncol[i]:=j;
      end;
    bcol:=j;
    for i:=1 to tlin+1 do
      begin
        act:=lac[i];
        while act^.pro<>pnil do
          begin
            bct:=act^.pro;
```

```
            FreeMem(act,6+tamreal*word(nlin[act^.blo]-nlin[act^.blo-1]));
            act:=bct;
          end;
        lac[i]:=lacf[i];
      end;
    at:=0;
    AtOt:=1;
    TmpFat:=TempoSis;
    Fatoracao;
    CalcX;
    nfat:=false;
    FatEcon:=1.0;
    TmpFat:=TempoSis-TmpFat+2*TmpBZY;
    TmpAtOt:=TmpFat;
    TmpAtAc:=TmpFat;
    TmpNorma:=0;
    MemDsp:=MemAvail-MemMin;
  end; {NovaFatoracao}

Procedure Ciclo(var ent,sai: integer);

  var
    i,j,t,ncabl,blent,blsai,tp1,tp2,tp3: integer;
    cent                               : single;

Function CalcNorma: boolean;

  var i,j,k,nct,col: integer;

  begin
    TmpNorma:=TempoSis;
    norma:=0.0;
    CalcX;
    for i:=1 to tcol do pi^[i]:=g^[i];
    for i:=1 to tcol do
      begin
        col:=cabas^[i];
        j:=1;
        while col>gcol[j] do inc(j);
        if col<ncan[j]
          then
            begin
              nct:=cest[j]-cest[j-1];
              dec(col,gcol[j-1]+(nlin[j-1]+1)*nct);
              for k:=nlin[j-1]+1 to nlin[j] do
                pi^[k]:=pi^[k]-a[j]^[k*nct+col]*vu^[i];
              col:=cabas^[i]-gcol[j-1]+cest[j-1];
              for k:=tlin+1 to tcol do
                pi^[k]:=pi^[k]-aest[k-tlin]^[col]*vu^[i];
            end
```

```
          else
            begin
              col:=afart^[col-cest[j]];
              pi^[nlin[j-1]+abs(col)]:=pi^[nlin[j-1]
                                    +abs(col)]-col/abs(col)*vu^[i];
            end;
        end;
      for i:=1 to tcol do
        if abs(pi^[i])>norma then norma:=abs(pi^[i]);
      if norma>zeronor
        then CalcNorma:=true
        else CalcNorma:=false;
      TmpNorma:=TempoSis-TmpNorma;
    end; {CalcNorma}

Function CalcTempo: boolean;

  var fator: single;

  begin
    TmpAtAc:=TmpAtAc+TmpAt+2*TmpBZY+ord((at mod novox)=0)*TmpNorma;
    fator:=TmpAtAc*AtOt/(TmpAtOt*(at+1));
    if fator<1.0
      then
        begin
          TmpAtOt:=TmpAtAc;
          AtOt:=at+1;
          if ((((at+1) mod novox)=0) and (fator>0.999))
            then CalcTempo:=true
            else CalcTempo:=false;
        end
      else
        if (fator<FatEcon) or ((at mod novox)=0)
          then CalcTempo:=false
          else CalcTempo:=true;
    FatEcon:=fator;
  end; {CalcTempo}

Function CalcMem: boolean;

  begin
    if (not vermem) or ((MemDsp>MemMin) and (maxeau>((ufim-ufimf)*2)))
      then CalcMem:=false
      else CalcMem:=true;
  end; {CalcMem}
```

```
Procedure QuemEntra; {criterio de Dantzig}

  var
    i,j,k,tt: integer;
    t       : single;

  begin
    ent:=0;
    cent:=-zeroent;


    for i:=1 to nbloc+1 do
      begin
        ncabl:=cest[i]-cest[i-1];
        tp1:=gcol[i-1];
        tp2:=nlin[i-1];
        tp3:=cest[i-1];
        for j:=tp1+1 to ncan[i]-1 do
          if fobas^[j] then
            begin
              t:=obj1^[j];
              for k:=tp2+1 to nlin[i] do
                t:=t-pi^[k]*a[i]^[(k-tp2-1)*ncabl+j-tp1];
              for k:=1 to nlest do
                t:=t-pi^[tlin+k]*aest[k]^[j-tp1+tp3];
              if (1-2*ord(lim^[j]))*t<cent then
                begin
                  ent:=j;
                  cent:=(1-2*ord(lim^[j]))*t;
                  blent:=i;
                end;
            end;
        tp3:=cest[i];
        for j:=ncan[i] to gcol[i] do
          if fobas^[j] then
            begin
              tt:=afart^[j-tp3];
              t:=(1-2*ord(lim^[j]))*(obj1^[j]-pi^[abs(tt)+tp2]*tt/abs(tt));
              if t<cent then
                begin
                  ent:=j;
                  cent:=t;
                  blent:=i;
                end;
            end;
      end;
  end; {QuemEntra}
```

```
Procedure QuemSai;

  var
    i       : integer;
    del,csai: single;
    saen    : boolean;

  begin
    sai:=0;
    blsai:=0;
    csai:=inf;
    if lim^[ent]
      then del:=-1.0
      else del:=1.0;
    for i:=1 to tcol do
      if abs(pi^[i])>zerosai then
        begin
          if (pi^[i]*del)>0.0
            then
              begin
                if (vu^[i]/abs(pi^[i]))<csai then
                  begin
                    sai:=i;
                    csai:=vu^[i]/abs(pi^[i]);
                  end;
              end
            else if ((lsup^[cabas^[i]]-vu^[i])/abs(pi^[i]))<csai then
                   begin
                     sai:=i;
                     csai:=(lsup^[cabas^[i]]-vu^[i])/abs(pi^[i]);
                   end;
        end;
    if (lsup^[ent]<csai)
      then
        begin
          lim^[ent]:=not lim^[ent];
          csai:=lsup^[ent];
          sai:=-1;
        end;
    if sai<>0 then
      begin
        for i:=1 to tcol do
          begin
            vu^[i]:=vu^[i]-del*csai*pi^[i];
            if vu^[i]<zerox then vu^[i]:=0.0;
          end;
        z:=z+cent*csai;
        if sai>0 then
          begin
            if vu^[sai]>(lsup^[cabas^[sai]]/2) then lim^[cabas^[sai]]:=true;
```

```
            vu^[sai]:=lsup^[ent]*ord(lim^[ent])+del*csai;
            if vu^[sai]<zerox then vu^[sai]:=0.0;
            lim^[ent]:=false;
            i:=cabas^[sai];
            while gcol[blsai]<i do inc(blsai);
          end;
      end;
  end; {QuemSai}

  begin
    if KeyPressed and (ReadKey=#3) then Saida(InterrupcaoUsuario);
    Inc(niter);
    if niter>maxit then Saida(AtingidoLimIteracoes);
    for i:=1 to tcol do g^[i]:=obj1^[cabas^[i]];
    BTZY(pi^,g^);
    QuemEntra;
    if ent<>0 then
      begin
        for i:=1 to nlin[blent-1] do g^[i]:=0.0;
        for i:=nlin[blent]+1 to tlin do g^[i]:=0.0;
        tp1:=gcol[blent-1];
        tp2:=nlin[blent-1];
        tp3:=cest[blent-1];
        if ent<ncan[blent]
          then
            begin
              if blent<=nbloc then
                begin
                  ncab1:=cest[blent]-tp3;
                  for i:=tp2+1 to nlin[blent] do
                    g^[i]:=a[blent]^[(i-tp2-1)*ncab1+ent-tp1];
                end;
              for i:=tlin+1 to tcol do
                g^[i]:=aest[i-tlin]^[ent-tp1+tp3];
            end
          else
            begin
              for i:=tp2+1 to nlin[blent] do g^[i]:=0.0;
              for i:=tlin+1 to tcol do g^[i]:=0.0;
              t:=afart^[ent-cest[blent]];
              g^[tp2+abs(t)]:=t/abs(t);
            end;
        BZY(pi^,g^);
        QuemSai;
        if sai>0 then
          begin
            fobas^[cabas^[sai]]:=true;
            fobas^[ent]:=false;
            if not nfat then Atualizacao(ent,sai,blent,blsai);
```

```
            if nfat
              then NovaFatoracao(ent,sai)
              else nfat:=(((at mod novox)=0) and CalcNorma)
                         or CalcTempo or CalcMem or (at=maxat);
          end;
      end;
  end; {Ciclo}

Begin {programa principal}
  LeDados;
  Escalamento;
  Objetivo1;
  NovaFatoracao(0,0);
  repeat Ciclo(ent,sai) until (ent=0) or (abs(z)<zeroz);
  CalcZ;
  if z>zeroz then Saida(NaoHaSolFactivel);
  Objetivo2;
  Repeat Ciclo(ent,sai) until (ent=0) or (sai=0);
  NovaFatoracao(0,0);
  if sai=0
    then Saida(SolIlimitada)
    else Saida(SolOtimaEncontrada);
End.
```

```
Program Simplex_do_capitulo_5(input,output);

Const
  maxblo          =   101;
  maxebl          =  3500;
  maxcol          =   160;
  maxcot          =  3000;
  maxlin          =    60;
  maxlit          =  3000;
  maxest          =    10;
  maxelU          = 16000;
  maxelL          = 16000;
  maxeaU          =  5000;
  inf     : single=  1e20;
  zeroa   : single=  1e-7;
  zerox   : single=  1e-5;
  zeroz   : single=  1e-5;
  zerob   : single=  1e-5;
  zerol   : single=  1e-5;
  zeroesc: single=  1e-5;
  zeroent: single=  1e-5;
  zerosai: single=  1e-5;
  zerosis: single=  1e-5;
  zerou   : single=  1e-7;
  zerolu  : single=  1e-8;
  zeropi  : single=  1e-6;
  zeroac  : single=  1e-6;
  zeronor: single=  1e-4;
  maxat           =    30;
  maxit           = 10000;
  novox           =    10;
  mu              =   1.0;
  tamreal         =     4;

Type
  xyp     = record
              x,y,p: integer;
            end;
  tyL     = array[1..maxelL] of single;
  tyU     = array[1..maxelU] of single;
```

```
  tyupos = array[1..maxeaU] of xyp;
  tycoi  = array[0..maxcot] of integer;
  tylir  = array[0..maxlit] of single;
  tycbb  = array[1..maxcol] of boolean;
  tycib  = array[1..maxcol] of integer;
  tycrb  = array[1..maxcol] of single;
  tylib  = array[1..maxlin] of integer;
  tylrb  = array[1..maxlin] of single;
  tydad  = array[1..maxebl] of single;
  tyLUe  = array[1..maxest,1..maxest] of single;
  tylei  = array[1..maxest] of integer;
  tyler  = array[1..maxest] of single;
  tyinbl = array[1..maxblo] of integer;
  tyblo  = record
             cod,ncol,nlin,ncor,ncca,ncfo,ncar: integer;
             prod                             : single;
             est                              : ^tylib;
             afart                            : ^tycib;
             fobas,lim                        : ^tycbb;
             linf,lsup,obj2,obj1              : ^tycrb;
             x,b,bmin,sca                     : ^tylrb;
             dad,lest                         : ^tydad;
           end;
  tymat  = array[1..maxblo] of ^tyblo;

Var
  nbloc,nlest,tcol,bcol,tlin,niter,
  UlBloE,UlVarE,at,ufim,ufimf,ent,sai:  integer;
  norma,ctobj,zinter,zinic,z          :  double;
  indblo                              :  tyinbl;
  matriz                              :  tymat;
  l                                   : ^tyL;
  u                                   : ^tyU;
  upos                                : ^tyUpos;
  lpos,cpos,cabas                     : ^tycoi;
  icnalt,icn                          :  tylei;
  esc                                 :  tyler;
  luest                               :  tyLUe;
  JaHaX,fasei,nfat                    :  boolean;
  pi,g,xtmp                           : ^tylir;

Procedure Escalamento;

  var
    i,j,k,l,bloco: integer;
    t            : double;

  begin
    for bloco:=1 to nbloc do
      with matriz[bloco]^ do
```

```
      begin
        k:=ncor+1;
        for i:=1 to nlin do
          begin
            t:=zeroesc;
            l:=(i-1)*ncor;
            for j:=l+1 to l+ncor do
              if abs(dad^[j])>t then t:=abs(dad^[j]);
            for j:=l+1 to l+ncor do
              if abs(dad^[j]/t)<zeroa
                then dad^[j]:=0.0
                else dad^[j]:=dad^[j]/t;
            b^[i]:=b^[i]/t;
            sca^[i]:=1.0/t;
            while (k<=ncca) and (abs(afart^[k-ncor])<i) do inc(k);
            if (abs(afart^[k-ncor])=i) and (k<=ncca) then
              begin
                linf^[k]:=linf^[k]/t;
                lsup^[k]:=lsup^[k]/t;
              end;
          end;
      end;
    with matriz[nbloc+1]^ do
      begin
        k:=ncor+1;
        for i:=1 to nlest do
          begin
            t:=zeroesc;
            for j:=1 to nbloc do
              with matriz[j]^ do
                if (est^[i]>0) and (prod>t) then t:=prod;
            b^[i]:=b^[i]/t;
            esc[i]:=1.0/t;
            while (k<=ncca) and (abs(afart^[k-ncor])<i) do inc(k);
            if (abs(afart^[k-ncor])=i) and (k<=ncca) then
              begin
                linf^[k]:=linf^[k]/t;
                lsup^[k]:=lsup^[k]/t;
              end;
          end;
      end;
    ctobj:=zeroesc;
    for bloco:=1 to nbloc do
      with matriz[bloco]^ do
        for i:=1 to ncor do
          if abs(obj2^[i])>ctobj then ctobj:=abs(obj2^[i]);
    for bloco:=1 to nbloc do
      with matriz[bloco]^ do
        for i:=1 to ncor do obj2^[i]:=obj2^[i]/ctobj;
  end; {Escalamento}
```

```
Procedure LeDados;

  begin
    {esta subrotina contem a leitura dos vetores que definem o problema}
    {e tambem a alocacao de memoria dos vetores utilizados no programa }
    pi^[0]:=1;
    cabas^[0]:=0;
    cabas^[tcol+1]:=31000;
    sai:=31000;
  end; {LeDados}

Procedure Fatoracao(prblo: integer);

  var
    pivo,linbl,uinbl,pinbl,pinbl2,gcol,
    lcont,ucont,pcont,econt,i,j,k,kk,s,ss,tt: integer;
    t,t1,t2                                 : single;

  begin
    lcont:=1;
    ucont:=1;
    pcont:=1;
    econt:=0;
    pinbl2:=1;
    gcol:=0;
    for i:=1 to nlest do
      for j:=1 to nlest do
        luest[i,j]:=0.0;
    for i:=1 to prblo-1 do
      with matriz[indblo[i]]^ do
        begin
          for j:=1 to ncol-nlin do
            begin
              icn[econt+j]:=icnalt[econt+j];
              ss:=cabas^[icn[econt+j]]-gcol;
              if ss<=ncar then
                begin
                  k:=1;
                  while (est^[k]<>ss) and (k<=nlest) do inc(k);
                  if k<=nlest then luest[k,econt+j]:=prod*esc[k];
                end;
            end;
          inc(econt,ncol-nlin);
          inc(pinbl2,ncol);
          inc(gcol,ncar);
          inc(lcont,(nlin*nlin+nlin) div 2);
          inc(ucont,ncol*nlin-((nlin*nlin+nlin) div 2));
          inc(pcont,nlin);
        end;
```

```
for i:=prblo to nbloc do
  with matriz[indblo[i]]^ do
    begin
      uinbl:=ucont;
      pinbl:=pcont;
      linbl:=lcont;
      for j:=pinbl to pinbl+ncol-1 do
        cpos^[j]:=pinbl2-pinbl+j;
      for j:=1 to nlin*nlest do lest^[j]:=0.0;
      for j:=1 to nlin do
        begin
          tt:=(j-1)*ncor;
          for k:=j to ncol do
            begin
              ss:=cabas^[cpos^[pinbl+k-1]]-gcol;
              if ss<=ncor
                then u^[ucont+k-j]:=dad^[tt+ss]
                else if abs(afart^[ss-ncor])=j
                       then u^[ucont+k-j]:=afart^[ss-ncor]/j
                       else u^[ucont+k-j]:=0.0;
            end;
          s:=uinbl-1;
          for k:=1 to j-1 do
            begin
              t:=l^[lpos^[pinbl+k-1]+j-k];
              if abs(t)>zerou then
                for kk:=j to ncol do
                  u^[ucont+kk-j]:=u^[ucont+kk-j]+u^[s+kk-k]*t;
              inc(s,ncol-k);
            end;
          pivo:=ucont;
          for k:=ucont+1 to ucont+ncol-j do
            if abs(u^[k])>abs(u^[pivo]) then pivo:=k;
          if abs(u^[pivo])<zerolu then saida(BaseSingular);
          s:=uinbl-1;
          for k:=1 to j-1 do
            begin
              kk:=pivo-ucont+s+j-k;
              t:=u^[kk];
              u^[kk]:=u^[s+j-k];
              u^[s+j-k]:=t;
              inc(s,ncol-k);
            end;
          t:=u^[pivo];
          u^[pivo]:=u^[ucont];
          s:=cpos^[pcont];
          cpos^[pcont]:=cpos^[pcont+pivo-ucont];
          cpos^[pcont+pivo-ucont]:=s;
          for k:=ucont to ucont+ncol-j-1 do
            u^[k]:=-u^[k+1]/t;
```

```
        l^[lcont]:=t;
        lpos^[pcont]:=lcont;
        kk:=cpos^[pinbl+j-1];
        ss:=cabas^[kk]-gcol;
        if ss<=ncor
          then
            begin
              for k:=j+1 to nlin do
                l^[lcont+k-j]:=dad^[(k-1)*ncor+ss];
              k:=1;
              while (est^[k]<>ss) and (k<=nlest) do inc(k);
              if k<=nlest then
                lest^[(k-1)*nlin+pcont-pinbl+1]:=prod*esc[k];
            end
          else for k:=j+1 to nlin do l^[lcont+k-j]:=0.0;
        s:=0;
        ss:=0;
        for k:=1 to j-1 do
          begin
            t:=u^[uinbl+ss+j-k-1];
            if abs(t)>zerou then
              begin
                for kk:=j+1 to nlin do
                  l^[lcont+kk-j]:=l^[lcont+kk-j]+t*l^[linbl+kk+s-1];
                for kk:=1 to nlest do
                  lest^[(kk-1)*nlin+pcont-pinbl+1]:=
                                        lest^[(kk-1)*nlin+pcont-pinbl+1]
                                        +t*lest^[(kk-1)*nlin+k];
              end;
            inc(s,nlin-k);
            inc(ss,ncol-k);
          end;
        inc(lcont,nlin-j+1);
        inc(pcont);
        inc(ucont,ncol-j);
      end;
    for j:=1 to ncol-nlin do
      begin
        icn[econt+j]:=cpos^[pcont+j-1];
        icnalt[econt+j]:=icn[econt+j];
        ss:=cabas^[icn[econt+j]]-gcol;
        if ss<=ncor then
          begin
            k:=1;
            while (est^[k]<>ss) and (k<=nlest) do inc(k);
            if k<=nlest then luest[k,econt+j]:=prod*esc[k];
          end;
      end;
    inc(econt,ncol-nlin);
    inc(pinbl2,ncol);
```

```
        inc(gcol,ncar);
      end;
    with matriz[nbloc+1]^ do
      for j:=1 to ncol do
        begin
          icn[econt+j]:=tcol-ncol+j;
          ss:=afart^[cabas^[j+tlin+econt]-bcol-ncor];
          luest[abs(ss),econt+j]:=ss/abs(ss);
        end;
    for j:=tlin+1 to tcol do
      begin
        cpos^[j]:=icn[j-tlin];
        icn[j-tlin]:=-j+tlin;
      end;
    ufim:=ucont-1;
    ufimf:=ufim;
    uinbl:=1;
    ucont:=1;
    pinbl;=1;
    for i:=1 to nbloc do
      with matriz[indblo[i]]^ do
        begin
          for j:=nlin+1 to ncol do
            begin
              s:=0;
              ss:=0;
              for k:=1 to nlin do
                begin
                  t:=u^[uinbl+ss+j-1-k];
                  if abs(t)>zerou then
                    for kk:=1 to nlest do
                      luest[kk,ucont]:=luest[kk,ucont]
                                       +t*lest^[(kk-1)*nlin+k];
                  inc(s,nlin-k);
                  inc(ss,ncol-k);
                end;
              inc(ucont);
            end;
          inc(pinbl,nlin);
          inc(uinbl,((nlin*nlin-nlin) div 2)+(ncol-nlin)*nlin);
        end;
    for j:=1 to nlest-1 do
      begin
        pivo:=j;
        for k:=j+1 to nlest do
          if abs(luest[j,k])>abs(luest[j,pivo]) then pivo:=k;
        t1:=luest[j,pivo];
        if pivo<>j then
          begin
            tt:=cpos^[tlin+j];
```

```
            cpos^[tlin+j]:=cpos^[tlin+pivo];
            cpos^[tlin+pivo]:=tt;
            tt:=icn[j];
            icn[j]:=icn[pivo];
            icn[pivo]:=tt;
            for k:=1 to nlest do
              begin
                t2:=luest[k,pivo];
                luest[k,pivo]:=luest[k,j];
                luest[k,j]:=t2;
              end;
          end;
        for k:=j+1 to nlest do
          begin
            t2:=-luest[j,k]/t1;
            luest[j,k]:=t2;
            if abs(t2)>zerou then
              for kk:=j+1 to nlest do
                luest[kk,k]:=luest[kk,k]+t2*luest[kk,j];
          end;
      end;
    for i:=1 to nlest do
      if icn[i]<0 then
        begin
          j:=i;
          tt:=-icn[i];
          while tt<>i do
            begin
              k:=tt;
              tt:=-icn[tt];
              icn[k]:=j;
              j:=k;
            end;
          icn[i]:=j;
        end;
  end; {Fatoracao}

Procedure BZY(var z,y: tylir; prblo,ulblo: integer);

  var vx: tylir;

  Procedure LXY(var vx,y: tylir);

    var
      i,j,k,m,nlibl: integer;
      t            : single;

    begin
      for i:=1 to tcol do vx[i]:=y[i];
      if prblo<=nbloc then
```

```
      begin
        nlibl:=0;
        for i:=1 to prblo-1 do
          inc(nlibl,matriz[indblo[i]]^.nlin);
        for i:=prblo to ulblo do
          with matriz[indblo[i]]^ do
            begin
              for j:=nlibl+1 to nlibl+nlin do
                begin
                  k:=lpos^[j];
                  vx[j]:=vx[j]/l^[k];
                  if abs(vx[j])>zerosis
                    then
                      begin
                        for m:=1 to nlin-j+nlibl do
                          vx[j+m]:=vx[j+m]-vx[j]*l^[k+m];
                        for m:=tlin+1 to tcol do
                          vx[m]:=vx[m]-vx[j]*
                                 lest^[(m-tlin-1)*nlin+j-nlibl];
                      end
                    else vx[j]:=0.0;
                end;
              inc(nlibl,nlin);
            end;
      end;
    for i:=1 to nlest do
      begin
        t:=vx[tlin+i];
        for j:=1 to i-1 do
          t:=t-vx[tlin+j]*luest[i,j];
        if abs(t)>zerosis
          then vx[tlin+i]:=t/luest[i,i]
          else vx[tlin+i]:=0.0;
      end;
  end; {LXY}

Procedure UZX(var z,vx: tylir);

  var
    i,j,k,ucont,uest,xinbl,ip: integer;
    ut                       : xyp;
    t                        : single;

  begin
    for i:=ufim downto ufimf+1 do
      begin
        ut:=upos^[i-ufimf];
        if ut.p<0
          then
            begin
```

```
            vx[ut.x]:=vx[ut.x]+vx[ut.y]*u^[i];
            if abs(vx[ut.x])<zerosis then vx[ut.x]:=0.0;
          end
        else
          begin
            vx[ut.y]:=vx[ut.y]+vx[ut.x]*u^[i];
            if abs(vx[ut.y])<zerosis then vx[ut.y]:=0.0;
          end;
      ip:=abs(ut.p);
      if ut.y<>ip then
        begin
          t:=vx[ip];
          vx[ip]:=vx[ut.y];
          vx[ut.y]:=t;
        end;
    end;
  for i:=nlest downto 1 do
    begin
      for j:=i+1 to nlest do
        vx[tlin+i]:=vx[tlin+i]+vx[tlin+j]*luest[i,j];
      if abs(vx[tlin+i])<zerosis then
        vx[tlin+i]:=0.0;
    end;
  ucont:=ufimf;
  uest:=nlest-matriz[nbloc+1]^.ncol;
  xinbl:=tlin;
  for i:=nbloc downto 1 do
    with matriz[indblo[i]]^ do
      if ((i>=prblo) and (i<=ulblo)) or (ncol<>nlin)
        then
          begin
            dec(uest,ncol-nlin);
            dec(xinbl,nlin);
            for j:=nlin downto 1 do
              begin
                dec(ucont,ncol-nlin);
                for k:=1 to ncol-nlin do
                  vx[xinbl+j]:=vx[xinbl+j]
                               +vx[tlin+icn[uest+k]]*u^[ucont+k];
                dec(ucont,nlin-j);
                for k:=j+1 to nlin do
                  vx[xinbl+j]:=vx[xinbl+j]+vx[xinbl+k]*u^[ucont+k-j];
                if abs(vx[xinbl+j])<zerosis then vx[xinbl+j]:=0.0;
              end;
          end
        else
          begin
            dec(xinbl,nlin);
            dec(ucont,(nlin*nlin-nlin) div 2);
          end;
```

```
      for i:=1 to tcol do
        z[cpos^[i]]:=vx[i];
    end; {UZX}

  begin
    LXY(vx,y);
    UZX(z,vx);
  end; {BZY}

Procedure UtXY(var vx,y: tylir; prblo: integer; JaHaX,GuardaX: boolean);

  var
    i,j,k,ip,xinbl,ucont,uest: integer;
    ut                       : xyp;
    t                        : single;

  begin
    for i:=1 to tcol do vx[i]:=y[cpos^[i]];
    xinbl:=0;
    ucont:=0;
    uest:=0;
    for i:=1 to prblo-1 do
      with matriz[indblo[i]]^ do
        begin
          inc(xinbl,nlin);
          inc(ucont,ncol*nlin-((nlin*nlin+nlin) div 2));
          inc(uest,ncol-nlin);
        end;
    if JaHaX then
      begin
        for i:=1 to xinbl do vx[i]:=xtmp^[i];
        for i:=1 to uest do vx[tlin+icn[i]]:=xtmp^[tlin+icn[i]];
      end;
    for i:=prblo to nbloc do
      with matriz[indblo[i]]^ do
        begin
          for j:=1 to nlin do
            begin
              t:=vx[xinbl+j];
              if abs(t)>zerosis
                then
                  begin
                    for k:=j+1 to nlin do
                      vx[xinbl+k]:=vx[xinbl+k]+t*u^[ucont+k-j];
                    inc(ucont,nlin-j);
                    for k:=1 to ncol-nlin do
                      vx[tlin+icn[uest+k]]:=vx[tlin+icn[uest+k]]
                                            +t*u^[ucont+k];
                    inc(ucont,ncol-nlin);
                  end
```

```
            else inc(ucont,ncol-j);
          end;
        inc(xinbl,nlin);
        inc(uest,ncol-nlin);
      end;
  if GuardaX then
    begin
      JaHaX:=true;
      for i:=1 to tlin-matriz[indblo[nbloc]]^.nlin do xtmp^[i]:=vx[i];
      for i:=tlin+1 to tcol do xtmp^[i]:=vx[i];
    end;
  for i:=1 to nlest-1 do
    begin
      t:=vx[tlin+i];
      if abs(t)>zerosis then
        for j:=i+1 to nlest do
          vx[tlin+j]:=vx[tlin+j]+t*luest[i,j];
    end;
  for i:=1 to tlin+nlest do
    if abs(vx[i])<zerosis then vx[i]:=0.0;
  for i:=ufimf+1 to ufim do
    begin
      ut:=upos^[i-ufimf];
      ip:=abs(ut.p);
      if ut.y<>ip then
        begin
          t:=vx[ip];
          vx[ip]:=vx[ut.y];
          vx[ut.y]:=t;
        end;
      if ut.p<0
        then
          begin
            vx[ut.y]:=vx[ut.y]+vx[ut.x]*u^[i];
            if abs(vx[ut.y])<zerosis then vx[ut.y]:=0.0;
          end
        else
          begin
            vx[ut.x]:=vx[ut.x]+vx[ut.y]*u^[i];
            if abs(vx[ut.x])<zerosis then vx[ut.x]:=0.0;
          end;
    end;
end; {UtXY}
```

```
Procedure LtZX(var z: tylir; var bloco,nlibl: integer);

  var
    j,k,m: integer;
    r    : single;

  begin
    with matriz[indblo[bloco]]^ do
      begin
        dec(nlibl,nlin);
        for j:=nlibl+nlin downto nlibl+1 do
          begin
            r:=0;
            for m:=tlin+1 to tcol do
              r:=r-z[m]*lest^[(m-tlin-1)*nlin+j-nlibl];
            k:=lpos^[j];
            for m:=1 to nlibl+nlin-j do
              r:=r-z[j+m]*l^[k+m];
            z[j]:=(z[j]+r)/l^[k];
            if abs(z[j])<zerosis then z[j]:=0.0;
          end;
      end;
  end; {LtZX}

Procedure BtZY(var z,y: tylir; JaHaX: boolean);

  var
    i,j: integer;
    r  : single;

  begin
    if JaHaX
      then i:=nbloc
      else i:=1;
    UtXY(z,y,i,JaHaX,not JaHaX);
    for i:=nlest downto 1 do
      begin
        r:=0;
        for j:=i+1 to nlest do
          r:=r-z[tlin+j]*luest[j,i];
        z[tlin+i]:=(z[tlin+i]+r)/luest[i,i];
        if abs(z[tlin+i])<zerosis then z[tlin+i]:=0.0;
      end;
  end; {BtZY}
```

```
Procedure Atualizacao(ent,sai,blent,blsai,glent,gcsai,gent,gsai: integer);

  var
    cpri,cult,i,j,ip,li,bl,bli,k,lp,s,it,ncabl,ss,si,st: integer;
    t,tt,gt                                            : single;
    act,bct                                            : tylrb;

  begin
    Inc(at);
    for i:=1 to tcol do g^[i]:=0;
    g^[gsai]:=1;
    UtXY(pi^,g^,nbloc,false,false);
    i:=tcol;
    while abs(pi^[i])<=zeropi do Dec(i);
    cult:=i;
    ip:=i-1;
    if cult<=tlin
      then
        begin
          bli:=nbloc;
          k:=lpos^[i];
          for j:=1 to i-glent-1 do act[j]:=0.0;
          for j:=i to tlin do
            act[j-glent]:=l^[k+j-i];
          with matriz[indbloc[nbloc]]^ do
            for j:=1 to nlest do bct[j]:=lest^[(j-1)*nlin+i-glent];
        end
      else
        begin
          bli:=nbloc+1;
          for j:=1 to tlin-glent do act[j]:=0.0;
          for j:=cult-tlin to nlest do
            bct[j]:=luest[j,cult-tlin];
          for j:=1 to cult-tlin-1 do bct[j]:=0.0;
          while ip>tlin do
            if abs(pi^[ip])<=zeropi
              then Dec(ip)
              else
                begin
                  Inc(ufim);
                  if abs(pi^[i])<abs(pi^[ip])
                    then
                      begin
                        t:=-pi^[i]/pi^[ip];
                        for j:=ip-tlin to i-tlin-1 do
                          begin
                            bct[j]:=luest[j,ip-tlin];
                            luest[j,ip-tlin]:=luest[j,ip-tlin]*t;
                          end;
                        for j:=i-tlin to nlest do
```

```
                    begin
                      tt:=luest[j,ip-tlin];
                      luest[j,ip-tlin]:=luest[j,ip-tlin]*t+bct[j];
                      bct[j]:=tt;
                    end;
                  upos^[ufim-ufimf].p:=ip;
                  i:=ip;
                end
              else
                begin
                  t:=-pi^[ip]/pi^[i];
                  upos^[ufim-ufimf].p:=cult;
                  for j:=i-tlin to nlest do
                    luest[j,ip-tlin]:=luest[j,ip-tlin]+bct[j]*t;
                end;
            u^[ufim]:=t;
            upos^[ufim-ufimf].x:=ip;
            upos^[ufim-ufimf].y:=cult;
            Dec(ip);
          end;
    end;
li:=i;
bl:=nbloc;
with matriz[indblo[nbloc]]^ do
  begin
    for k:=ip downto glent+1 do
      if abs(pi^[k])>zeropi then
        begin
          Inc(ufim);
          lp:=lpos^[k]-k;
          if abs(pi^[li])<abs(pi^[k])
            then
              begin
                t:=-pi^[li]/pi^[k];
                upos^[ufim-ufimf].p:=k;
                if bl<bli
                  then
                    begin
                      for j:=k to tlin do
                        begin
                          act[j-glent]:=l^[lp+j];
                          l^[lp+j]:=l^[lp+j]*t;
                        end;
                      for j:=1 to k-glent-1 do act[j]:=0.0;
                    end
                  else
                    begin
                      for j:=k to li-1 do
                        begin
                          act[j-glent]:=l^[lp+j];
```

```
                    l^[lp+j]:=l^[lp+j]*t;
                  end;
                for j:=li to tlin do
                  begin
                    tt:=l^[lp+j];
                    l^[lp+j]:=act[j-glent]+t*tt;
                    act[j-glent]:=tt;
                  end;
              end;
            for j:=1 to nlest do
              begin
                tt:=lest^[(j-1)*nlin+k-glent];
                lest^[(j-1)*nlin+k-glent]:=bct[j];
                bct[j]:=tt;
              end;
            li:=k;
            bli:=bl;
          end
        else
          begin
            t:=-pi^[k]/pi^[li];
            upos^[ufim-ufimf].p:=cult;
            if bl=bli then
              for j:=li to nlin+glent do
                l^[lp+j]:=l^[lp+j]+t*act[j-glent];
          end;
      u^[ufim]:=t;
      upos^[ufim-ufimf].x:=k;
      upos^[ufim-ufimf].y:=cult;
      for j:=1 to nlest do
       lest^[(j-1)*nlin+k-glent]:=lest^[(j-1)*nlin+k-glent]+bct[j]*t;
    end;
  gt:=pi^[li];
  for j:=1 to 2 do
    begin
      if j=1
        then
          begin
            ss:=blent;
            st:=ent;
            si:=1;
          end
        else
          begin
            ss:=blsai;
            st:=cabas^[gsai]-gcsai;
            si:=-1;
          end;
      cpri:=glent+1;
      if st<=ncor
```

```
        then
          for i:=1 to nlin do
            act[i]:=act[i]+gt*si*dad^[(i-1)*ncor+st]
        else
          begin
            i:=afart^[st-ncor];
            act[abs(i)]:=act[abs(i)]+gt*si*i/abs(i);
          end
    end;
  for j:=2 downto 1 do
    begin
      if j=1 then
        begin
          ss:=blent;
          si:=1;
          st:=ent;
        end;
      if st<=ncor then
        for i:=1 to nlest do
          if est^[i]=st then
            bct[i]:=bct[i]+gt*si*prod*esc[i];
    end;
  cabas^[gsai]:=gent;
  if cult>tlin
    then it:=tlin
    else it:=cult-1;
  i:=cpri;
  while i<=it do
    begin
      if abs(act[i-glent])>zeroac then
        begin
          Inc(ufim);
          s:=lpos^[i];
          if abs(l^[s])*mu>abs(act[i-glent])
            then
              begin
                t:=-act[i-glent]/l^[s];
                upos^[ufim-ufimf].p:=-cult;
                for j:=i+1 to tlin do
                  act[j-glent]:=act[j-glent]+l^[s+j-i]*t;
              end
            else
              begin
                t:=-l^[s]/act[i-glent];
                upos^[ufim-ufimf].p:=-i;
                for j:=i to tlin do
                  begin
                    tt:=act[j-glent];
                    act[j-glent]:=l^[s+j-i]+tt*t;
                    l^[s+j-i]:=tt;
```

```
                end;
              for j:=1 to nlest do
                begin
                  tt:=bct[j];
                  bct[j]:=lest^[(j-1)*nlin+i-glent];
                  lest^[(j-1)*nlin+i-glent]:=tt;
                end;
            end;
          u^[ufim]:=t;
          upos^[ufim-ufimf].x:=i;
          upos^[ufim-ufimf].y:=cult;
          for j:=1 to nlest do
            bct[j]:=bct[j]+lest^[(j-1)*nlin+i-glent]*t;
        end;
      inc(i);
    end;
  if cult>tlin
    then
      begin
        it:=1;
        for i:=it to cult-tlin-1 do
          if abs(bct[i])>zeroac then
            begin
              Inc(ufim);
              if abs(luest[i,i])*mu>abs(bct[i])
                then
                  begin
                    t:=-bct[i]/luest[i,i];
                    upos^[ufim-ufimf].p:=-cult;
                    for j:=i+1 to nlest do
                      bct[j]:=bct[j]+luest[j,i]*t;
                  end
                else
                  begin
                    t:=-luest[i,i]/bct[i];
                    upos^[ufim-ufimf].p:=-i-tlin;
                    for j:=i to nlest do
                      begin
                        tt:=bct[j];
                        bct[j]:=luest[j,i]+tt*t;
                        luest[j,i]:=tt;
                      end;
                  end;
                u^[ufim]:=t;
                upos^[ufim-ufimf].x:=i+tlin;
                upos^[ufim-ufimf].y:=cult;
              end;
        for i:=cult-tlin to nlest do
          luest[i,cult-tlin]:=bct[i];
      end
```

```
          else
            begin
              s:=lpos^[cult];
              for i:=cult to tlin do
                l^[s+i-cult]:=act[i-glent];
              for i:=1 to nlest do
                lest^[(i-1)*nlin+cult-glent]:=bct[i];
            end;
        end;
    end; {Atualizacao}

Procedure Calc_Z(total: boolean);

  var i,nlibl,ncibl: integer;

  Procedure ZBlo(bloco: integer);

    var j: integer;

    begin
      with matriz[indblo[bloco]]^ do
        begin
          for j:=1 to ncca do
            if lim^[j] then z:=z+lsup^[j]*obj1^[j];
          for j:=1 to ncol do
            z:=z+x^[j]*obj1^[cabas^[nlibl+j]-ncibl];
          inc(nlibl,ncol);
          inc(ncibl,ncar);
        end;
    end; {ZBlo}

  begin
    if total
      then z:=zinic
      else z:=zinter;
    nlibl:=0;
    ncibl:=0;
    if total
      then
        begin
          for i:=1 to nbloc-1 do zblo(i);
          zinter:=z;
        end
      else
        for i:=1 to nbloc-1 do
          with matriz[indblo[i]]^ do
            begin
              inc(nlibl,ncol);
              inc(ncibl,ncar);
            end;
```

```
    for i:=nbloc to nbloc+1 do zblo[i];
  end; {Calc_Z}

Procedure Calc_X(total: boolean);

  var
    i,j,nlibl,t: integer;
    tt         : single;

  begin
    nlibl:=tcol;
    for i:=nbloc+1 downto 1 do
      with matriz[indblo[i]]^ do
        begin
          dec(nlibl,nlin);
          for j:=1 to nlin do
            g^[j+nlibl]:=b^[j];
          for j:=1 to ncor do
            if lim^[j] then
              begin
                tt:=lsup^[j];
                for t:=1 to nlin do
                  g^[t+nlibl]:=g^[t+nlibl]-tt*dad^[(t-1)*ncor+j];
                for t:=1 to nlest do
                  if est^[t]=j then
                    g^[tlin+t]:=g^[tlin+t]-tt*prod*esc[t];
              end;
          for j:=ncor+1 to ncca do
            if lim^[j] then
              begin
                t:=afast^[j-ncor];
                g^[nlibl+abs(t)]:=g^[nlibl+abs(t)]-lsup^[j]*t/abs(t);
              end;
        end;
    DZY(pi^,g^,1,nblo);
    nlibl:=0;
    for i:=1 to nbloc+1 do
      with matriz[indblo[i]]^ do
        begin
          for j:=1 to ncol do
            x^[j]:=-pi^[nlibl+j];
          inc(nlibl,ncol);
        end;
    Calc_Z(total);
  end; {Calc_X}
```

```
Function CalcNorma: boolean;

  var j,k,col,nlibl,ncibl: integer;

  begin
    norma:=0.0;
    Calc_X(false);
    nlibl:=0;
    ncibl:=0;
    for j:=1 to nbloc-1 do
      begin
        inc(nlibl,matriz[indbloc[j]]^.ncol);
        inc(ncibl,matriz[indbloc[j]]^.ncar);
      end;
    for j:=tlin-matriz[indbloc[nbloc]]^.nlin+1 to tlin do pi^[j]:=g^[j];
    with matriz[indbloc[nbloc]]^ do
      for j:=1 to ncol do
        begin
          col:=cabas^[j+nlibl]-ncibl;
          if col<=ncor
            then
              begin
                for k:=1 to nlin do
                  pi^[k+nlibl]:=pi^[k+nlibl]-dad^[(k-1)*ncor+col]*x^[j];
                for k:=1 to nlest do
                  if est^[k]=col then
                    pi^[tlin+k]:=pi^[tlin+k]-prod*esc[k]*x^[j];
              end
            else
              begin
                col:=afart^[col-ncor];
                pi^[nlibl+abs(col)]:=pi^[nlibl+abs(col)]-col/abs(col)*x^[j];
              end;
        end;
    for j:=tlin-matriz[indbloc[nbloc]]^.nlin+1 to tlin do
      if abs(pi^[j])>norma then norma:=abs(pi^[j]);
    if norma>zeronor
      then CalcNorma:=true
      else CalcNorma:=false;
  end; {CalcNorma}

Procedure Objetivo1;

  var i,j,k,kk,ncibl: integer;

  begin
    fase1:=true;
    niter:=0;
    zinic:=0;
    k:=0;
```

```
    ncibl:=0;
    for i:=1 to nbloc+1 do
      with matriz[i]^ do
        begin
          kk:=k;
          for j:=1 to ncca do
            begin
              obj1^[j]:=0.0;
              fobas^[j]:=true;
              lim^[j]:=false;
            end;
          for j:=ncca+1 to ncfo do
            begin
              if afart^[j-ncor]>0
                then
                  begin
                    inc(k);
                    cabas^[k]:=j+ncibl;
                    fobas^[j]:=false;
                  end
                else fobas^[j]:=true;
              lim^[j]:=false;
              obj1^[j]:=0.0;
            end;
          for j:=ncfo+1 to ncar do
            begin
              obj1^[j]:=1;
              fobas^[j]:=false;
              inc(k);
              cabas^[k]:=j+ncibl;
              lim^[j]:=false;
            end;
          inc(ncibl,ncar);
          ncol:=k-kk;
        end;
    for i:=1 to nbloc+1 do indblo[i]:=i;
  end; {Objetivo1}

Procedure Objetivo2;

  var i,j: integer;

  begin
    niter:=0;
    fase1:=false;
    sai:=31000;
    for i:=1 to nbloc+1 do
      with matriz[i]^ do
        begin
          for j:=ncfo+1 to ncar do
```

```
            begin;
              fobas^[j]:=false;
              obj1^[j]:=0.0;
            end;
          for j:=1 to ncor do
            begin
              obj1^[j]:=obj2^[j];
              zinic:=zinic+obj1^[j]*linf^[j];
            end;
        end;
    Calc_Z(true);
  end; {Objetivo2}

Procedure OrdenaCabas(pcol: integer); {InsertionSort}

  var i,j,v: integer;

  begin
    for i:=pcol to tcol do
      begin
        v:=cabas^[i];
        j:=i;
        while cabas^[j-1]>v do
          begin
            cabas^[j]:=cabas^[j-1];
            Dec(j);
          end;
        cabas^[j]:=v;
      end;
  end; {OrdenaCabas}

Procedure MudaOrdem(bloco: integer);

  var
    i,j1,j2,k,kp1,kp2,difc,ncibl1,ncibl2: integer;
    cabaux                              : tycib;

  begin
    j1:=0;
    ncibl1:=0;
    for i:=1 to bloco-1 do
      with matriz[indblo[i]]^ do
        begin
          inc(ncibl1,ncar);
          inc(j1,ncol);
        end;
    j2:=j1;
    ncibl2:=ncibl1;
    for i:=bloco to nbloc-1 do
      with matriz[indblo[i]]^ do
```

```
        begin
          inc(ncibl2,ncar);
          inc(j2,ncol);
        end;
    difc:=matriz[indblo[nbloc]]^.ncar-matriz[indblo[bloco]]^.ncar;
    k:=matriz[indblo[nbloc]]^.ncol-matriz[indblo[bloco]]^.ncol;
    if k>0
      then
        begin
          with matriz[indblo[nbloc]]^ do
            for i:=1 to ncol do cabaux[i]:=cabas^[j2+i]-ncibl2+ncibl1;
          kp2:=j2+k;
          with matriz[indblo[bloco]]^ do
            for i:=1 to ncol do
              cabas^[kp2+i]:=cabas^[j1+i]-ncibl1+ncibl2+difc;
          kp1:=j1+matriz[indblo[bloco]]^.ncol+1;
          for i:=j2 downto kp1 do
            cabas^[i+k]:=cabas^[i]+difc;
          with matriz[indblo[nbloc]]^ do
            for i:=1 to ncol do
              cabas^[j1+i]:=cabaux[i];
        end
      else
        begin
          with matriz[indblo[bloco]]^ do
            for i:=1 to ncol do cabaux[i]:=cabas^[j1+i]-ncibl1+ncibl2+difc;
          kp1:=j1+matriz[indblo[bloco]]^.ncol+1;
          with matriz[indblo[nbloc]]^ do
            for i:=1 to ncol do
              cabas^[j1+i]:=cabas^[j2+i]-ncibl2+ncibl1;
          kp2:=j2+k;
          for i:=kp1 to j2 do
            cabas^[i+k]:=cabas^[i]+difc;
          with matriz[indblo[bloco]]^ do
            for i:=1 to ncol do
              cabas^[kp2+i]:=cabaux[i];
        end;
    i:=indblo[bloco];
    indblo[bloco]:=indblo[nbloc];
    indblo[nbloc]:=i;
  end; {MudaOrdem}

Procedure NovaFatoracao(gent,gsai,blent,blsai: integer);

  var i,j,blin: integer;

  begin
    nfat:=false;
    cabas^[gsai]:=gent;
    j:=0;
```

```
    if blent<blsai
      then blin:=blent
      else blin:=blsai;
    if blin>nbloc then blin:=nbloc;
    for i:=1 to blin-1 do inc(j,matriz[indbloc[i]]^.ncol);
    OrdenaCabas(j+1);
    if blent<>blsai then
      begin
        inc(matriz[indbloc[blent]]^.ncol);
        dec(matriz[indbloc[blsai]]^.ncol);
      end;
    if blent<nbloc then MudaOrdem(blent);
    JaHaX:=false;
    at:=0;
    Fatoracao(blin);
    if blin=nbloc
      then Calc_X(false)
      else Calc_X(true);
  end; {NovaFatoracao}

Procedure Ciclo(var ent,sai: integer);

  var
    i,j,t,nlibl,ncibl,blent,blsai,glent,gent,gsai,gcsai: integer;
    cent                                               : single;

Procedure QuemEntra; {criterio de Dantzig com filtro}

  var bloco: integer;

  Procedure SelColuna(bloco: integer);

    var
      j,k,tt: integer;
      t     : single;

    begin
      with matriz[indbloc[bloco]]^ do
        begin
          for j:=1 to ncor do
            if fobas^[j] then
              begin
                t:=obji^[j];
                for k:=1 to nlin do
                  t:=t-pi^[nlibl+k]*dad^[(k-1)*ncor+j];
                for k:=1 to nlest do
                  if est^[k]=j then
                    t:=t-pi^[tlin+k]*prod*esc[k];
                if (1-2*ord(lim^[j]))*t<cent then
                  begin
```

```
                    ent:=j;
                    cent:=(1-2*ord(lim^[j]))*t;
                    blent:=bloco;
                  end;
              end;
          for j:=ncor+1 to ncar do
            if fobas^[j] then
              begin
                tt:=afart^[j-ncor];
                t:=(1-2*ord(lim^[j]))
                   *(obj1^[j]-pi^[abs(tt)+nlibl]*tt/abs(tt));
                if t<cent then
                  begin
                    ent:=j;
                    cent:=t;
                    blent:=bloco;
                  end;
              end;
        end;
    end; {SelColuna}

  begin
    ent:=0;
    cent:=-zeroent;
    nlibl:=tlin;
    SelColuna(nbloc+1);
    bloco:=nbloc;
    repeat
      LtZX(pi^,bloco,nlibl);
      SelColuna(bloco);
      dec(bloco);
    until ((cent<(-zeroent*100.0)) and ((UlVarE<>ent) or (blent<>UlBloE)))
          or (bloco=0);
    UlBloE:=blent;
    UlVarE:=ent;
    glent:=0;
    gent:=ent;
    for bloco:=1 to blent-1 do
      with matriz[indblo[bloco]]^ do
        begin
          inc(gent,ncar);
          inc(glent,nlin);
        end;
  end; {QuemEntra}
```

```
Procedure QuemSai;

  var
    i,bloco : integer;
    del,csai: single;
    saen    : boolean;

  Procedure Atualiza_X(csai: single);

    var bloco,i: integer;

    begin
      nlibl:=0;
      for bloco:=1 to nbloc+1 do
        with matriz[indblo[bloco]]^ do
          begin
            for i:=1 to ncol do
              begin
                x^[i]:=x^[i]-del*csai*pi^[i+nlibl];
                if abs(x^[i])<zerox then x^[i]:=0.0;
              end;
             inc(nlibl,ncol);
          end;
    end; {Atualiza_X}

  begin
    sai:=0;
    blsai:=0;
    csai:=inf;
    if matriz[indblo[blent]]^.lim^[ent]
      then del:=-1
      else del:=1;
    nlibl:=0;
    ncibl:=0;
    for bloco:=1 to nbloc+1 do
      with matriz[indblo[bloco]]^ do
        begin
          for i:=1 to ncol do
            if abs(pi^[nlibl+i])>zerosai then
                begin
                  if (pi^[nlibl+i]*del)>zerosai
                    then
                      begin
                        if (x^[i]/abs(pi^[nlibl+i]))<csai then
                          begin
                            sai:=i;
                            csai:=x^[i]/abs(pi^[nlibl+i]);
                            blsai:=bloco;
                          end;
                      end
```

```
                else if ((lsup^[cabas^[i+nlibl]-ncibl]
                        -x^[i])/abs(pi^[nlibl+i]))<csai then
                      begin
                        sai:=i;
                        csai:=(lsup^[cabas^[i+nlibl]-ncibl]
                              -x^[i])/abs(pi^[nlibl+i]);
                        blsai:=bloco;
                      end;
            end;
      inc(nlibl,ncol);
      inc(ncibl,ncar);
    end;
with matriz[indblo[blent]]^ do
  if lsup^[ent]<csai
    then
      begin
        lim^[ent]:=not lim^[ent];
        z:=z+cent*lsup^[ent];
        sai:=-1;
        Atualiza_X(lsup^[ent]);
      end;
if sai>0 then
  begin
    z:=z+cent*csai;
    gsai:=sai;
    gcsai:=0;
    for i:=1 to blsai-1 do
      with matriz[indblo[i]]^ do
        begin
          inc(gsai,ncol);
          inc(gcsai,ncar);
        end;
    if ((blent=nbloc) and (blent=blsai) and (at<maxat))
      then
        begin
          Atualiza_X(csai);
          with matriz[indblo[blsai]]^ do
            begin
              if x^[sai]>(lsup^[cabas^[gsai]-gcsai]/2) then
                lim^[cabas^[gsai]-gcsai]:=true;
              x^[sai]:=lsup^[ent]*ord(lim^[ent])+del*csai;
              if abs(x^[sai])<zerox then x^[sai]:=0.0;
            end;
        end
      else
        begin
          nfat:=true;
          with matriz[indblo[blsai]]^ do
            if (x^[sai]-del*csai*pi^[gsai])>(lsup^[cabas^[gsai]-gcsai]/2)
              then lim^[cabas^[gsai]-gcsai]:=true;
```

```
            end;
          matriz[indblo[blent]]^.lim^[ent]:=false;
        end;
    end; {QuemSai}


    begin
      Inc(niter);
      if niter>maxit then Saida(AtingidoLimIteracoes);
      nlibl:=0;
      ncibl:=0;
      for i:=1 to nbloc+1 do
        with matriz[indblo[i]]^ do
          begin
            for j:=nlibl+1 to nlibl+ncol do
              g^[j]:=objl^[cabas^[j]-ncibl];
            inc(nlibl,ncol);
            inc(ncibl,ncar);
          end;
      BTZY(pi^,g^,JaHaX);
      QuemEntra;
      if ent<>0 then
        begin
          for i:=1 to tcol do g^[i]:=0.0;
          with matriz[indblo[blent]]^ do
            if ent<=ncor
              then
                begin
                  for i:=1 to nlin do
                    g^[glent+i]:=dad^[(i-1)*ncor+ent];
                  for i:=1 to nlest do
                    if est^[i]=ent then g^[i+tlin]:=prod*esc[i];
                end
              else
                begin
                  t:=afart^[ent-ncor];
                  g^[glent+abs(t)]:=t/abs(t);
                end;
          BZY(pi^,g^,blent,blent);
          QuemSai;
          if sai>0 then
            begin
              matriz[indblo[blsai]]^.fobas^[cabas^[gsai]-gcsai]:=true;
              matriz[indblo[blent]]^.fobas^[ent]:=false;
              if nfat
                then NovaFatoracao(gent,gsai,blent,blsai)
                else
                  begin
                    Atualizacao(ent,sai,blent,blsai,glent,gcsai,gent,gsai);
                    nfat:=(((at mod novox)=0) and CalcNorma);
```

```
                end
            end;
        end;
    end; {Ciclo}

Begin {programa principal}
  LeDados;
  Escalamento;
  Objetivo1;
  NovaFatoracao(0,0,1,1);
  repeat Ciclo(ent,sai) until (ent=0) or (abs(z)<zeroz);
  Calc_Z(false);
  if z>zeroz then Saida(NaoHaSolFactivel);
  Objetivo2;
  Repeat Ciclo(ent,sai) until (ent=0) or (sai=0);
  NovaFatoracao(0,0,1,1);
  if sai=0
    then Saida(SolIlimitada)
    else Saida(SolOtimaEncontrada);
End.
```

BIBLIOGRAFIA

[1] AMARAL, Luiz Roberto S. Um algoritmo estável para resolução do problema de otimização de rações. Tese de mestrado, Departamento de Matemática Aplicada, Universidade Estadual de Campinas, 1988.

[2] BARTELS, Richard H. A stabilization of the simplex method. *Numerische Mathematik*, Berlin, Springer-Verlag, **16** (5):414-434, 1971.

[3] BARTELS, Richard H. & GOLUB, Gene H. The simplex method of linear programming using LU decomposition. *Communications of the ACM*, New York, **12** (5):266-268, may 1969.

[4] ——— Algorithm 350: simplex method procedure employing LU decomposition. *Communications of the ACM*, New York, **12** (5):275-278, may 1969.

[5] BARTELS, Richard H. et alii. A realization of the simplex method based on triangular decompositions, contribution I/11. in: WILKINSON, J. H. & REINSCH, C., eds. *Handbook for automatic computation, vol II: linear algebra.* Berlin, Springer-Verlag, 1971. p. 152-190.

[6] BENICHOU, M. et alii. The efficient solution of large-scale linear programming problems: some algorithmic techniques and computational results. *Mathematical Programming*, Amsterdam, North-Holland, **13** (3):280-322, dec.1977.

[7] DUFF, I. S.; ERISMAN, A. M.; REID, J. K. *Direct methods for sparse matrices.* Oxford, Clarendon Press, 1986. 341p.

[8] FORSYTHE, G. E.; MALCOM, M. A.; MOLER, C. B. *Computer methods for mathematical computations*. Englewood-Cliffs, Prentice-Hall, 1977. p. 48-55.

[9] GILL, Philip E.; MURRAY, Walter; SAUDERS, Michel A.; WRIGHT, Margaret H. Mantaining LU factors of a general sparse matrix. Stanford, *Technical Report SOL 86-8*, Department of Operation Research, Stanford University, may 1986. 28p.

[10] GILL, Philip E.; MURRAY, Walter; WRIGHT, Margaret H. *Practical optimization*. London, Academic Press, 1981. 401p.

[11] KENNINGTON, Jeff L. & HELGASON, Richard V. *Algorithms for network programming*. New york, John Wiley, 1980. 191p.

[12] LASDON, Leon S. *Optimization theory for large systems*. New York, Macmillan, 1970. 523p.

[13] LUENBERGER, David G. *Linear and nonlinear programming*, 2. ed. Reading, Mass., Addison-Wesley, 1984. 491 p.

[14] REID, J. K. A note on the stability of gaussian elimination. *Journal of the Institute of Mathematics and Its Applications*. London, Academic Press, 8 (3):374-375, dec. 1971.

[15] ROSEN, J. B. Primal partition programming for block diagonal matrices. *Numerische Mathematik*, Berlin, Springer-Verlag, 6 (3):250-260, aug. 1964.

ÍNDICE

CAPÍTULO SEGUNDO

CAPÍTULO TERCEIRO