
Computational Techniques of the Simplex Method

**INTERNATIONAL SERIES IN
OPERATIONS RESEARCH & MANAGEMENT SCIENCE**
Frederick S. Hillier, Series Editor

Stanford University

- Gal, T., Stewart, T.J., Hanne, T. / *MULTICRITERIA DECISION MAKING: Advances in MCDM Models, Algorithms, Theory, and Applications*
- Fox, B.L. / *STRATEGIES FOR QUASI-MONTE CARLO*
- Hall, R.W. / *HANDBOOK OF TRANSPORTATION SCIENCE*
- Grassman, W.K. / *COMPUTATIONAL PROBABILITY*
- Pomerol, J-C. & Barba-Romero, S. / *MULTICRITERION DECISION IN MANAGEMENT*
- Axsäter, S. / *INVENTORY CONTROL*
- Wolkowicz, H., Saigal, R., & Vandenberghe, L. / *HANDBOOK OF SEMI-DEFINITE PROGRAMMING: Theory, Algorithms, and Applications*
- Hobbs, B.F. & Meier, P. / *ENERGY DECISIONS AND THE ENVIRONMENT: A Guide to the Use of Multicriteria Methods*
- Dar-El, E. / *HUMAN LEARNING: From Learning Curves to Learning Organizations*
- Armstrong, J.S. / *PRINCIPLES OF FORECASTING: A Handbook for Researchers and Practitioners*
- Balsamo, S., Personé, V., & Onvural, R. / *ANALYSIS OF QUEUEING NETWORKS WITH BLOCKING*
- Bouyssou, D. et al. / *EVALUATION AND DECISION MODELS: A Critical Perspective*
- Hanne, T. / *INTELLIGENT STRATEGIES FOR META MULTIPLE CRITERIA DECISION MAKING*
- Saaty, T. & Vargas, L. / *MODELS, METHODS, CONCEPTS and APPLICATIONS OF THE ANALYTIC HIERARCHY PROCESS*
- Chatterjee, K. & Samuelson, W. / *GAME THEORY AND BUSINESS APPLICATIONS*
- Hobbs, B. et al. / *THE NEXT GENERATION OF ELECTRIC POWER UNIT COMMITMENT MODELS*
- Vanderbei, R.J. / *LINEAR PROGRAMMING: Foundations and Extensions, 2nd Ed.*
- Kimms, A. / *MATHEMATICAL PROGRAMMING AND FINANCIAL OBJECTIVES FOR SCHEDULING PROJECTS*
- Baptiste, P., Le Pape, C. & Nuijten, W. / *CONSTRAINT-BASED SCHEDULING*
- Feinberg, E. & Shwartz, A. / *HANDBOOK OF MARKOV DECISION PROCESSES: Methods and Applications*
- Ramík, J. & Vlach, M. / *GENERALIZED CONCAVITY IN FUZZY OPTIMIZATION AND DECISION ANALYSIS*
- Song, J. & Yao, D. / *SUPPLY CHAIN STRUCTURES: Coordination, Information and Optimization*
- Kozan, E. & Ohuchi, A. / *OPERATIONS RESEARCH/ MANAGEMENT SCIENCE AT WORK*
- Bouyssou et al. / *AIDING DECISIONS WITH MULTIPLE CRITERIA: Essays in Honor of Bernard Roy*
- Cox, Louis Anthony, Jr. / *RISK ANALYSIS: Foundations, Models and Methods*
- Dror, M., L'Ecuyer, P. & Szidarovszky, F. / *MODELING UNCERTAINTY: An Examination of Stochastic Theory, Methods, and Applications*
- Dokuchaev, N. / *DYNAMIC PORTFOLIO STRATEGIES: Quantitative Methods and Empirical Rules for Incomplete Information*
- Sarker, R., Mohammadian, M. & Yao, X. / *EVOLUTIONARY OPTIMIZATION*
- Demeulemeester, R. & Herroelen, W. / *PROJECT SCHEDULING: A Research Handbook*
- Gazis, D.C. / *TRAFFIC THEORY*
- Zhu, J. / *QUANTITATIVE MODELS FOR PERFORMANCE EVALUATION AND BENCHMARKING*
- Ehrgott, M. & Gandibleux, X. / *MULTIPLE CRITERIA OPTIMIZATION: State of the Art Annotated Bibliographical Surveys*
- Bienstock, D. / *Potential Function Methods for Approx. Solving Linear Programming Problems*
- Matsatsinis, N.F. & Siskos, Y. / *INTELLIGENT SUPPORT SYSTEMS FOR MARKETING DECISIONS*
- Alpern, S. & Gal, S. / *THE THEORY OF SEARCH GAMES AND RENDEZVOUS*
- Hall, R.W. / *HANDBOOK OF TRANSPORTATION SCIENCE - 2nd Ed.*
- Glover, F. & Kochenberger, G.A. / *HANDBOOK OF METAHEURISTICS*
- Graves, S.B. & Ringuest, J.L. / *MODELS AND METHODS FOR PROJECT SELECTION: Concepts from Management Science, Finance and Information Technology*
- Hassin, R. & Haviv, M. / *TO QUEUE OR NOT TO QUEUE: Equilibrium Behavior in Queueing Systems*
- Gershwin, S.B. et al / *ANALYSIS & MODELING OF MANUFACTURING SYSTEMS*

COMPUTATIONAL TECHNIQUES OF THE SIMPLEX METHOD

ISTVÁN MAROS
Imperial College, London
and
Computer and Automation Research Institute, Budapest

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available from the Library of Congress.

Maros, I./ COMPUTATIONAL TECHNIQUES OF THE SIMPLEX METHOD

ISBN 978-1-4613-4990-7 ISBN 978-1-4615-0257-9 (eBook)

DOI 10.1007/978-1-4615-0257-9

Copyright © 2003 by Springer Science+Business Media New York

Originally published by Kluwer Academic Publishers in 2003

Softcover reprint of the hardcover 1st edition 2003

All rights reserved. No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording, or otherwise, without the written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

Printed on acid-free paper.

**To my wife Anna and
children Anikó and
András**

Contents

Preface	xvii
---------	------

Part I Preliminaries

1. THE LINEAR PROGRAMMING PROBLEM	3
1.1 Standard form	3
1.2 General form	4
1.2.1 Computational form #1	5
1.2.2 Computational form #2	13
1.3 Concluding remarks	17
1.3.1 Translation of bounds	17
1.3.2 Yet another formulation	17
2. THE SIMPLEX METHOD	19
2.1 Theoretical background	20
2.1.1 Basis and solution	21
2.1.2 The geometry of constraints	23
2.1.3 Neighboring bases	26
2.2 The primal simplex method	28
2.2.1 Optimality conditions	28
2.2.2 Improving a nonoptimal basic feasible solution	29
2.2.3 Algorithmic description of the simplex method	33
2.2.4 Finding a basic feasible solution	34
2.2.5 The two-phase simplex method	37
2.3 Duality	38
2.4 The dual simplex method	40
2.4.1 Dual feasibility	40

2.4.2	Improving a dual feasible solution	41
2.4.3	Algorithmic steps of the dual	43
2.4.4	Further properties of the primal-dual relationship	46
2.5	Concluding remarks	47
3.	LARGE SCALE LP PROBLEMS	49
3.1	Sparsity	49
3.2	Instances of large problems	51
3.3	Structure	52
3.4	Fill-in	53
3.5	Numerical difficulties	55
 Part II Computational Techniques		
4.	DESIGN PRINCIPLES OF LP SYSTEMS	59
4.1	Requirements of LP software	60
4.2	Computer hardware	62
4.3	The software side of implementing LP solvers	65
5.	DATA STRUCTURES AND BASIC OPERATIONS	69
5.1	Storing sparse vectors	70
5.2	Operations involving sparse vectors	71
5.2.1	Addition and accumulation of sparse vectors	71
5.2.2	Dot product of sparse vectors	74
5.3	Storing sparse matrices	74
5.4	Linked lists	76
5.5	Implementation of forward/backward linking	80
5.6	Concluding remarks	84
6.	PROBLEM DEFINITION	87
6.1	The MPS format	87
6.2	Processing the MPS format	93
7.	LP PREPROCESSING	97
7.1	Presolve	97
7.1.1	Contradicting individual bounds	100
7.1.2	Empty rows	100
7.1.3	Empty columns	100
7.1.4	Singleton rows	100

7.1.5	Removing fixed variables	101
7.1.6	Redundant and forcing constraints	102
7.1.7	Tightening individual bounds	102
7.1.8	Implied free variables	104
7.1.9	Singleton columns	105
7.1.10	Dominated variables	105
7.1.11	Reducing the sparsity of A	107
7.1.12	The algorithm	108
7.1.13	Implementation	109
7.2	Scaling	110
7.3	Postsolve	116
7.3.1	Unscaling	117
7.3.2	Undo presolve	118
7.3.3	Undo reformulation	119
8.	BASIS INVERSE, FACTORIZATION	121
8.1	Product form of the inverse	122
8.1.1	General form	123
8.1.2	Sparsity exploiting form	124
8.1.3	Implementing the PFI	129
8.1.4	Operations with the PFI	131
8.2	LU factorization	134
8.2.1	Determining a sparse LU form	136
8.2.2	Implementing the LU factorization	143
8.2.3	Maintaining triangularity during iterations	149
8.2.4	Operations with the LU form	155
8.3	Concluding remarks	158
9.	THE PRIMAL ALGORITHM	161
9.1	Solution, feasibility, infeasibility	162
9.2	Optimality conditions	164
9.3	Improving a BFS	167
9.3.1	The logic of the ratio test	173
9.3.2	Extensions to the ratio test	175
9.3.3	Algorithmic description of PSM-G	176
9.3.4	Computational considerations	178
9.4	Determining the reduced costs	181
9.4.1	Computing d_j	182
9.4.2	Updating π	182

9.4.3	Updating d_j	183
9.5	Selecting an incoming (improving) variable	184
9.5.1	Dantzig pricing	187
9.5.2	Partial pricing	187
9.5.3	Sectional pricing	188
9.5.4	Normalized pricing	189
9.5.5	A general pricing framework	198
9.6	Improving an infeasible basic solution	203
9.6.1	Analysis of $w(t)$	208
9.6.2	The logic of the ratio test	217
9.6.3	Extensions to the ratio test	217
9.6.4	Computational considerations	224
9.6.5	Adaptive composite pricing in phase-1	227
9.7	Handling degeneracy	230
9.7.1	Anti-degeneracy column selection	232
9.7.2	Wolfe's 'ad hoc' method	234
9.7.3	Expanding tolerance	237
9.7.4	Perturbation, shifting	241
9.8	Starting bases	244
9.8.1	Logical basis	244
9.8.2	Crash bases	246
9.8.3	CPLEX basis	253
9.8.4	A tearing algorithm	255
9.8.5	Partial basis	259
10.	THE DUAL ALGORITHM	261
10.1	Dual feasibility, infeasibility	261
10.2	Improving a dual feasible solution	262
10.2.1	Dual algorithm with type-1 and type-2 variables	263
10.2.2	Dual algorithm with all types of variables	266
10.2.3	Bound flip in dual	267
10.2.4	Extended General Dual algorithm	270
10.3	Improving a dual infeasible solution	277
10.3.1	Analysis of the dual infeasibility function	281
10.3.2	Dual phase-1 iteration with all types of variables	284
10.4	Row selection (dual pricing)	292
10.4.1	Dantzig pricing	293
10.4.2	Steepest edge	293

10.4.3	Devex	296
10.4.4	Pricing in dual phase-1	297
10.5	Computing the pivot row	297
10.6	Degeneracy in the dual	298
11.	VARIOUS ISSUES	301
11.1	Run parameters	301
11.1.1	Parameter file	302
11.1.2	String parameters	302
11.1.3	Algorithmic parameters	303
11.1.4	Numerical parameters	303
11.2	Performance evaluation, profiling	303
11.3	Some alternative techniques	306
11.3.1	Superbasic variables	306
11.3.2	Row basis	307
11.4	Modeling systems	310
11.5	A prototype simplex solver	311
	Index	321

Foreword

The first efficient solution method of the linear programming problem, the simplex method, was created by G.B. Dantzig in 1947. We celebrated the 50th anniversary of its birth at the Lausanne Symposium on Mathematical Programming in 1997. This is the tool that made the linear programming problem numerically solvable and widely accepted by the scientific community. Other methods also emerged, in the course of the time, most significant are among them the interior point methods, but Dantzig's approach survives as the main tool to solve not only practical but also theoretical problems in the field. Soon after its discovery it became obvious that the number of arithmetic operations needed to solve numerically the problem is extremely large and we will not succeed without high speed computational tools. Fortunately, ENIAC, the first electronic computer was patented just a year before the birth of the simplex method. Still, it was not in a shape that it could directly be used to solve LP's and this deficiency became, later on, the driving force in the development of electronic computers.

Simultaneously, mathematicians gave the simplex method other, computationally more tractable forms, and in about 15 years professional LP solvers became widely available. Much of the activity in this respect is associated with the name of Orchard-Hays, who, in 1968, published his landmark book [Orchard-Hays, 1968] on advanced linear programming computing techniques.

While in the USA there was a great interest in linear programming, right from the beginning, other countries joined the efforts later. Operations research started in Hungary in 1957. As the result of the 1956 revolution, literature on the application of mathematics in economics (and also other sciences) became more and more available. When we encountered the first books and articles on linear programming, we became delighted by its beauty. The fact that convex polyhedra in higher

dimensional spaces can accurately describe various production, scheduling, transportation, mixing and other problems and finitely convergent algorithms can produce numerically the extreme values of linear functions on those sets, fascinated us. Research in and teaching of linear programming and its extensions began. By about 1960 I already had a regular graduate level course on linear programming, given to students in applied mathematics, at the Lornd Etvs University of Budapest. Istvn Maros became a student in my class of 1962. Istvn had already built a solid mathematical as well as computational background in himself. The latter one was not frequent among students of mathematics. He, however, was strong in his interest and scientific devotion and went along to study and do research in the numerical direction. Shortly after graduation from the University of Budapest, he became a national expert in the numerical solution of linear programming problems, including those of large scale. By about 1970 he succeeded to solve a large scale LP, of decomposition type, formulated for an economic planning problem. He used the methods, presented in the book of Orchard-Hays, in a creative manner.

When I stepped down from the position of head of the Department of Applied Mathematics of the Computing and Automation Research Institute of the Hungarian Academy of Sciences, I chose Istvn as my successor. He continued his research but this time he worked on a PC. He developed an extremely efficient implementation of the simplex method, which, later on, came out as second best in an international comparison conducted by R. Sharda (EJOR, 1989). Istvn, however, made his work alone in the academia, while the code that became the first, was a professional one, developed by a group. He had a number of other successes, too. For example, in 1980 ICL invited him to London, to build in his LP solver, called MILP, into an animal feed optimization system. They had chosen MILP because it was the fastest among all candidates in the given framework. So, the author of the present book is not only an expert of the theory of the subject he is dealing with but also excellent in its practical realization. In the book Istvn Maros is publishing now, he discloses his secrets and presents them to the interested readers. The book is, however, not only about that. It presents a comprehensive description of those algorithmic elements which are needed for a state-of-the-art implementation of the simplex method, in a novel and attractive setting. Several of the results are of his own.

The main feature of the book is the presentation of a huge richness of algorithms and algorithmic components not seen in any other book on the simplex method so far. The author combines them with some known data structure methods in an intuitive way. Of particular im-

portance are the use of piecewise linear objective functions in phase-1 and phase-2 of the dual algorithms that not only make the dual a perfect alternative to the primal but also boosts the performance of mixed integer programming algorithms by greatly reducing the effort needed for reoptimization in branch and bound/cut. All algorithmic elements presented in the book have been thoroughly tested.

The book can be used by specialists who develop LP solver codes customized for special purposes. It can also be used as a textbook, to teach graduate course, as well as a monograph helping to do research on the subject. It is clearly written, easy to read and deserves to be on the bookshelf of every mathematical programmer.

András Prékopa

Member of the Hungarian Academy of Sciences

Preface

Without any doubt, linear programming (LP) is the most frequently used optimization technique. One of the reasons for its popularity is the fact that very powerful solution algorithms exist for linear optimization. Computer programs based on either on the simplex or interior point methods are capable of solving very large-scale problems with high reliability and within reasonable time. Model builders are aware of this and often try to formulate real-life problems within this framework to ensure they can be solved efficiently. Efficiency is obviously an important feature since it is typical that many different versions of a problem need to be solved in a practical application.

It is also true that many real-life optimization problems can be formulated as truly linear models and also many others can well be approximated by linearization. Furthermore, the need for solving LP problems may arise during the solution of other optimization problems, like mixed integer linear programming (MILP).

The two main methods for solving LP problems are the variants of the simplex method (SM) and the interior point methods (IPMs). They were in fierce competition in the 1990s to claim superiority but the battle—from practical point of view—is over now. It has turned out that both variants have their role in solving different problems.

It is interesting to note that while IPM was the newcomer in the mid 1980s it reached maturity both in theoretical and computational sense within one and a half decades. All aspects of it are well documented in [Terlaky, 1996] and [Roos et al., 1997]. The simplex method, however, seems to be an evergreen area for researchers. In the 1970s it was thought to have been matured but the appearance of IPMs had a stimulating effect on the research on SM. It is commonly accepted that, as a result, the efficiency of simplex based solvers has increased by two orders of magnitude within this period. It has been due to the developments

of the (*i*) theoretical background of optimization algorithms, (*ii*) inclusion of relevant results of computer science, (*iii*) using the principles of software engineering, and (*iv*) taking into account the state-of-the-art in computer technology. This list shows how many different aspects are to be aware of when trying to prepare a high quality optimization software.

The simplex method has been a favorite topic of the literature of operations research. Over the years a number of excellent books and papers have been published concentrating on various aspects of it. Without attempting to give a comprehensive list, I just mention a few, [Hadley, 1962, Dantzig, 1963, Orchard-Hays, 1968, Beale, 1968, Murtagh, 1981, Chvátal, 1983, Nazareth, 1987, Padberg, 1995] and [Prékopa, 1996]. There are other textbooks that cover the simplex method as part of a broader scope, like [Hillier and Lieberman, 2000] and [Vanderbei, 2001]. The wider community has never been able to enjoy the unique book of Prékopa [Prékopa, 1968] which, unfortunately, has not been translated into English.

If there is such a great coverage of the topic then why to write one more book on the simplex method?

It is well known that a theoretically correct algorithm can be implemented in many different ways. However, the performance of the different versions will vary enormously, including—correctly coded—variants that are unable to solve even some small problems. It can occur also with the simplex method. Therefore, it is of great importance to know how to implement it. It is not simply about coding. All sorts of new algorithms and algorithmic elements have contributed to the steady improvement of the simplex based optimizers. Actually, the success is based on the proper synthesis of the above mentioned (*i*)–(*iv*) components. I am one of those who have been working on these issues within the academia. Over the years I was approached by several people asking for advice regarding the implementation of the simplex method. They were complaining that information was scarce and scattered in the literature. Additionally, they were not aware of all the aspects to be considered. This and several other factors have persuaded me to try to fill this evident gap by preparing a book devoted to the computational issues of the simplex method. My main intention is to provide sufficient details about all aspects that are necessary for a successful implementation of the simplex method. It is impossible to cover everything. Therefore, I decided to concentrate on the most important parts and give them an in-depth treatment. I believe, on the basis of the material of the book the reader, with reasonable programming background, will be able to prepare a high quality implementation.

The book is divided into two parts. Part I covers the theoretical background and the basic versions of the primal and dual simplex method (aspect (i) above). It also gives an analysis of large-scale LP problems that serves as a motivation for the advanced issues. The main feature of such problems is sparsity, i.e., a low percentage of the nonzero elements in the matrix of constraints.

Part II contains the real essence of the book. First, it presents the general design principles of optimization software, then data structures for sparse problems and the basic operations on them, followed by the discussion of the industry standard MPS format for specifying an LP problem (aspects (ii)–(iv)). Large-scale LP models typically contain redundancies. If they are eliminated smaller, but equivalent, problems are solved. Heuristic methods have been developed to detect and eliminate redundancies and bring the problems into a form more suitable for solution. This is part of the preprocessing which is discussed next. The importance of improving the numerical characteristics of the problems has long been known. It usually can be achieved by scaling which is also part of the preprocessing. Inversion and/or LU factorization of the basis matrix plays a key role in the simplex method. This topic is presented in great detail. Also the efficient organization of the operations needed to solve some linear equations in every iteration is given full coverage. The discussion of the computational versions of the primal method is quite comprehensive. All aspects of the iterations are investigated and alternative algorithmic techniques are presented. In a simplistic way it can be said that it is all about determining the incoming and outgoing variables. The currently known best phase-1 and phase-2 methods are detailed here. Similar treatment is given to the dual simplex method. It is shown that the dual, endowed with an efficient phase-1 algorithm, is a completely realistic alternative to the primal. In many cases it even performs much better. The last chapter of the book discusses some practical, not necessarily algorithmic, details of the implementation. Also, some possible algorithmic alternatives are outlined here. Finally, a summary is given explaining the structure and main requirements of the implementation of the simplex method. All the latter topics relate to aspects (i)–(iii) but, when relevant, (iv) is also given due consideration.

I would like to acknowledge the encouragement and help I have received since the inception of the idea of this book. Most of them are due to András Prékopa, Fred Hillier (the series editor), Gautam Mitra, Ibrahim Osman, Berç Rustem and Sampo Ruuth. I found my years at the Department of Computing of Imperial College very inspirational which also contributed to the successful completion of the book.

Fred Hillier, the series editor, and Gary Folven at Kluwer were very patient when I was late with the delivery of the camera-ready manuscript. Many thanks for their understanding and continual support.

Last but not least I would like to express my utmost appreciation for the warm support and love I have received from my family. Their important role cannot be compared to any other factor. This is why I dedicate this book to them.

ISTVÁN MAROS

I

PRELIMINARIES

Chapter 1

THE LINEAR PROGRAMMING PROBLEM

In verbal terms, the linear programming problem (also referred to as linear program, LP) is to optimize (minimize or maximize) a linear function subject to linear equality and/or inequality constraints.

The original version of the LP problem as presented by Dantzig in [Dantzig, 1963] is considered the classical form. It is well known that any LP problem can be converted into this form if appropriate equivalent transformations are carried out. From practical, and also theoretical, point of view, however, it is important to investigate the most general form arising in practice and create algorithmic techniques within the framework of the simplex method that not only can cope with such problems but can also take advantage of them.

First we briefly present the classical standard form of the LP problem then discuss the general form in detail.

1.1. Standard form

The standard form of the LP problem is to find the minimum (or maximum) of a linear function of \bar{n} *nonnegative variables*, $x_1, x_2, \dots, x_{\bar{n}}$, also called *decision variables*, subject to m linear equality constraints. Formally,

$$\text{minimize} \quad c_1x_1 + \cdots + c_{\bar{n}}x_{\bar{n}} \quad (1.1)$$

$$\text{subject to} \quad a_1^i x_1 + \cdots + a_{\bar{n}}^i x_{\bar{n}} = b_i, \quad i = 1, \dots, m \quad (1.2)$$

$$x_j \geq 0, \quad j = 1, \dots, \bar{n}. \quad (1.3)$$

where c_j , a constant associated with activity (variable) j , is often called *cost coefficient*, a_j^i is the (technological) coefficient of variable j in con-

straint i , and b_i is the right-hand side (RHS) coefficient of constraint i . (1.1) is called *objective function*, (1.2) is the set of *joint constraints* (each involving several variables), and (1.3) is referred to as *nonnegativity restrictions*. In this formulation all constraints are equalities and all variables are restricted to be nonnegative. Using matrix notation, (1.1) – (1.3) can be written as:

$$\text{minimize} \quad \mathbf{c}^T \mathbf{x} \quad (1.4)$$

$$\text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b} \quad (1.5)$$

$$\mathbf{x} \geq \mathbf{0}, \quad (1.6)$$

where $\mathbf{A} \in \mathbb{R}^{m \times \bar{n}}$, $\mathbf{c}, \mathbf{x} \in \mathbb{R}^{\bar{n}}$ and $\mathbf{b} \in \mathbb{R}^m$.

It is customary to denote the value of the objective function by z . Therefore, we also will use notation $z = \mathbf{c}^T \mathbf{x} = c_1 x_1 + \cdots + c_{\bar{n}} x_{\bar{n}}$.

It can be shown that other traditionally used forms, where the joint constraints are allowed to be equalities or inequalities, can be converted into the standard form above. Readers interested in the equivalence and conversions are referred to some standard textbooks on linear programming. The only purpose of presenting the (1.4) – (1.6) standard form here is to enable the introduction of the basic concepts of the simplex method.

1.2. General form

During model formulation we encounter all sorts of constraints and variables not just the ones present in the standard form. Additionally, the objective function may also be slightly different as it can have an additive constant c_0 (the starting value)

$$z = c_0 + c_1 x_1 + \cdots + c_{\bar{n}} x_{\bar{n}} \quad \text{or} \quad z = c_0 + \sum_{j=1}^{\bar{n}} c_j x_j \quad (1.7)$$

The general constraints that involve more than one variable can have lower and upper bounds in the following way:

$$L_i \leq \sum_{j=1}^{\bar{n}} a_j^i x_j \leq U_i, \quad i = 1, \dots, m. \quad (1.8)$$

Individual constraints on variables are often referred to as bounds and in the general form they look like:

$$\ell_j \leq x_j \leq u_j \quad j = 1, \dots, \bar{n}. \quad (1.9)$$

Any of the lower bounds (L_i or ℓ_j) can be $-\infty$. Similarly, any of the upper bounds (U_i or u_j) can be $+\infty$.

The structure of the constraints and bounds can be visualized if we put them together in a tabular form:

	x_1	\dots	x_j	\dots	$x_{\bar{n}}$	
L_1	a_1^1	\dots	a_j^1	\dots	$a_{\bar{n}}^1$	U_1
\vdots	\vdots		\vdots		\vdots	\vdots
L_i	a_1^i	\dots	a_j^i	\dots	$a_{\bar{n}}^i$	U_i
\vdots	\vdots		\vdots		\vdots	\vdots
L_m	a_1^m	\dots	a_j^m	\dots	$a_{\bar{n}}^m$	U_m
ℓ_1	1					u_1
\vdots						\vdots
ℓ_j			1			u_j
\vdots						\vdots
$\ell_{\bar{n}}$					1	$u_{\bar{n}}$

There are several special cases of the general form based on the finiteness of one or both bounds of a constraint or a variable. Depending on how these cases are handled, we can develop several different forms that are mathematically equivalent to the general form of (1.7) – (1.9). We give details of two alternative formulations. Each of them is suitable for implementing advanced algorithmic and computational techniques of the simplex method.

It is to be noted that we do not consider the inclusion of the objective function in the constraint matrix. While it would give a bit more uniformity to the discussion, it would reduce the clarity of the description of some algorithmic elements and also the flexibility of adding new features to the solution algorithm. Furthermore, it is computationally not even advantageous.

1.2.1 Computational form #1

This form follows some ideas and formulation first put forward by Orchard-Hays [Orchard-Hays, 1968] but is not identical with it.

1.2.1.1 Individual bounds

- 1 If $u_j = +\infty$ and ℓ_j is finite then (1.9) can simply be written as $\ell_j \leq x_j$. Such an x_j is called a *plus type or nonnegative variable* and will be referred to by label PL.
- 2 If $\ell_j = -\infty$ and u_j is finite then (1.9) becomes $x_j \leq u_j$. Such an x_j is called a *minus type variable*. Its label is MI. Minus type variables can be converted into plus type by substituting $\bar{x}_j = -x_j$ and $\bar{\ell}_j = -u_j$ leading to $\bar{\ell}_j \leq \bar{x}_j$. This substitution can be performed during preprocessing and it leads to a reasonable reduction in the number of different types of variables.
- 3 If both ℓ_j and u_j are finite, an x_j satisfying (1.9), $\ell_j \leq x_j \leq u_j$, is called a *bounded variable* and label BD is assigned to it. There is a special sub-case here when $\ell_j = x_j = u_j$. Such an x_j is called a *fixed variable* and its label is FX. From practical point of view, a fixed variable is not a variable but a constant that could be excluded from the problem. While solvers essentially take these variables out and restore them after a solution has been obtained, it is important to provide facilities for algorithmic handling of fixed variables. They may emerge naturally, for example, during the solution of (mixed) integer programming problems by branch and bound as a result of fixing a variable.
- 4 If $\ell_j = -\infty$ and $u_j = +\infty$ then (1.9) essentially becomes $-\infty \leq x_j \leq +\infty$ and such an x_j is called an *unrestricted or free variable*. Its label is FR. Free variables can also be written as a difference of two nonnegative variables $x_j = x_j^+ - x_j^-$, where $x_j^+, x_j^- \geq 0$. If this substitution is used then matrix \mathbf{A} is augmented by a column since $\mathbf{a}_j x_j = \mathbf{a}_j x_j^+ - \mathbf{a}_j x_j^-$ and the new column vector corresponding to x_j^- is $-\mathbf{a}_j$.

1.2.1.2 General constraints

Now we discuss the noteworthy special cases of the joint constraints and introduce a variable for each constraint to turn them into equalities. constraint type

- 1 If $L_i = -\infty$ and U_i is finite then (1.8) becomes $\sum_{j=1}^n a_j^i x_j \leq U_i$. Using notation $b_i = U_i$ we get $\sum_{j=1}^n a_j^i x_j \leq b_i$. Such a constraint is referred to as *\leq type constraint* and label LE is assigned to it. By adding a nonnegative variable z_i to this constraint, it can be

converted into an equality:

$$z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = b_i, \quad z_i \geq 0.$$

- 2 If $U_i = +\infty$ and L_i is finite then (1.8) can be written as $L_i \leq \sum_{j=1}^{\bar{n}} a_j^i x_j$. Such a constraint is better known as $\sum_{j=1}^{\bar{n}} a_j^i x_j \geq L_i$ and is called \geq type constraint. Its label is GE. Denoting $b_i = -L_i$ we obtain the equivalent form: $\sum_{j=1}^{\bar{n}} (-a_j^i) x_j \leq b_i$. This, again, can be converted into an equation by adding a nonnegative variable z_i

$$z_i + \sum_{j=1}^{\bar{n}} (-a_j^i) x_j = b_i, \quad z_i \geq 0.$$

- 3 If both L_i and U_i are finite then (1.8) corresponds to two general constraints $L_i \leq \sum_{j=1}^{\bar{n}} a_j^i x_j$ and $\sum_{j=1}^{\bar{n}} a_j^i x_j \leq U_i$. Obviously, they are equivalent to a general constraint and an individual bound: $z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = U_i$ with $0 \leq z_i \leq (U_i - L_i)$. This is called a range constraint. Label RG is assigned to it. Denoting $b_i = U_i$ and $R_i = (U_i - L_i)$ we obtain

$$z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = b_i, \quad 0 \leq z_i \leq R_i.$$

It is true even if $L_i = U_i$ in which case $z_i = 0$, giving one more example of the importance of enabling the inclusion of fixed variables. This special case is referred to as equality constraint and its label is EQ.

- 4 If $L_i = -\infty$ and $U_i = +\infty$ then with arbitrary b_i , we can formally write

$$z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = b_i, \quad z_i \text{ is an unrestricted (free) variable.}$$

This case is referred to as a non-binding constraint and is labelled NB. Arguably, the name is somewhat misleading. If it is non-binding then why call it a constraint? Even more, why keep it in the model at all? The answer to the latter is that such 'constraints' may arise naturally during model formulation (e.g., inclusion of several objective functions for evaluation in multi-objective optimization), or when simplex is used as a subroutine. Other times the difference between

the left hand side and a given (target) value (b_i) is of specific interest. Non-binding constraints can be eliminated from the problem during preprocessing and restored at the end when the solution is presented.

From the above discussion we can conclude that with an appropriately chosen z_i variable any general constraint can be brought into the following form:

$$z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = b_i, \quad i = 1, \dots, m. \quad (1.10)$$

The z_i , $i = 1, \dots, m$, variables are called *logical variables* and x_j , $j = 1, \dots, \bar{n}$, are referred to as *structural variables*.

The rules of converting any general constraint into an equality can be summarized as follows:

- 1 Add a nonnegative logical variable z_i to each LE constraint.
- 2 Multiply each \geq type constraint by -1 (RHS element included) and add a nonnegative logical variable z_i (conversion to LE).
- 3 Set the RHS of a range constraint to the upper limit ($b_i = U_i$) and add a bounded logical variable $0 \leq z_i \leq R_i = U_i - L_i$ to the constraint.
- 4 Add a free logical variable $-\infty \leq z_i \leq +\infty$ to each non-binding constraint.
- 5 Finally, for uniformity: Add a zero valued logical variable $z_i = 0$ to each equality constraint.

This procedure is illustrated by the following example. Relational symbol $><$ refers to non-binding constraints.

EXAMPLE 1.1 Convert the following constraints into equalities

$$\begin{array}{rcl}
 & 3x_1 + 2x_2 + 6x_3 - x_4 \leq 9 \\
 & x_1 - x_2 + x_4 \geq 3 \\
 2 \leq & x_1 + 2x_2 + 3x_3 - 4x_4 \leq 15 \\
 & 2x_1 - x_2 - x_3 + x_4 >< 10 \\
 & x_1 + 3x_2 - x_3 + 6x_4 = 8 \\
 & x_1 \geq 0, x_2 \leq 0, -4 \leq x_3 \leq -1, x_4 \text{ free}
 \end{array}$$

Constraint-1: add nonnegative logical variable z_1 (part 1 of the rule)

$$z_1 + 3x_1 + 2x_2 + 6x_3 - x_4 = 9, \quad z_1 \geq 0$$

Constraint-2: multiply constraint by -1 and add a nonnegative logical variable z_2 (part 2 of the rule)

$$z_2 - x_1 + x_2 - x_4 = -3, \quad z_2 \geq 0$$

Constraint-3: set $b_i = 15$ and add a bounded logical variable $0 \leq z_3 \leq 15 - 2 = 13$ (part 3 of the rule)

$$z_3 + x_1 + 2x_2 + 3x_3 - 4x_4 = 15, \quad 0 \leq z_3 \leq 13$$

Constraint-4: add a free logical variable z_4 (part 4 of the rule)

$$z_4 + 2x_1 - x_2 - x_3 + x_4 = 10, \quad z_4 \text{ free}$$

Constraint-5: add a zero valued logical variable z_5 (part 5 of the rule)

$$z_5 + x_1 + 3x_2 - x_3 + 6x_4 = 8, \quad z_5 = 0$$

The resulting new set of constraints is

$$\begin{array}{rccccccccc} z_1 & + & 3x_1 & + & 2x_2 & + & 6x_3 & - & x_4 & = & 9 \\ z_2 & - & x_1 & + & x_2 & & & - & x_4 & = & -3 \\ z_3 & + & x_1 & + & 2x_2 & + & 3x_3 & - & 4x_4 & = & 15 \\ z_4 & + & 2x_1 & - & x_2 & - & x_3 & + & x_4 & = & 10 \\ z_5 & + & x_1 & + & 3x_2 & - & x_3 & + & 6x_4 & = & 8 \\ & & & & & & z_1, z_2 \geq 0, 0 \leq z_3 \leq 13, z_4 \text{ free}, z_5 = 0 \\ & & & & & & x_1 \geq 0, x_2 \leq 0, -4 \leq x_3 \leq -1, x_4 \text{ free} \end{array}$$

Using vector notation, (1.10) can be written as

$$\sum_{i=1}^m \mathbf{e}_i z_i + \sum_{j=1}^n \mathbf{a}_j x_j = \mathbf{b}. \quad (1.11)$$

The same with matrix notation is

$$\mathbf{I}\mathbf{z} + \mathbf{A}\mathbf{x} = \mathbf{b}.$$

Now we turn our attention to the individual bounds and make some reasonable changes in order to reduce the number of the cases for the solution algorithm.

Reversing MI variables. For convenience, MI variables are converted into PL. If we multiply $-\infty \leq x_j \leq u_j$ by -1 we get

$$-u_j \leq -x_j \leq +\infty.$$

Denoting $\bar{\ell}_j = -u_j$ and $\bar{x}_j = -x_j$, relation $\bar{\ell}_j \leq \bar{x}_j \leq +\infty$ is obtained where \bar{x}_j is a PL variable. Because of $x_j = -\bar{x}_j$, all the a_j^i coefficients in column j have to be multiplied by -1 . It should

be recorded that column j now represents \bar{x}_j . When a solution is reached \bar{x}_j has to be converted back to x_j by changing its sign.

Lower bound translation. All finite individual lower bounds can be moved to zero by *translation* (also known as *shift*) in the following way.

If $-\infty < \ell_k \leq x_k$ then let $\bar{x}_k = x_k - \ell_k$. It follows that $\bar{x}_k \geq 0$ and $x_k = \bar{x}_k + \ell_k$. If u_k is finite, it also changes $\bar{u}_k = u_k - \ell_k$. Substituting $x_k = \bar{x}_k + \ell_k$ into (1.11),

$$\sum_{i=1}^m \mathbf{e}_i z_i + \sum_{j \neq k} \mathbf{a}_j x_j + \mathbf{a}_k (\bar{x}_k + \ell_k) = \mathbf{b}, \quad (1.12)$$

which is further equal to

$$\sum_{i=1}^m \mathbf{e}_i z_i + \sum_{j \neq k} \mathbf{a}_j x_j + \mathbf{a}_k \bar{x}_k = \mathbf{b} - \mathbf{a}_k \ell_k. \quad (1.13)$$

Verbally: A finite lower bound ℓ_k of variable x_k can be moved to zero by subtracting ℓ_k times column k from the right-hand side. Column k of \mathbf{A} remains unchanged but now represents variable \bar{x}_k . When a solution is obtained x_k must be computed from $x_k = \bar{x}_k + \ell_k$.

EXAMPLE 1.2 In Example 1.1, from $-4 \leq x_3 \leq -1$, the finite lower bound of x_3 is different from zero. Introducing $\bar{x}_3 = x_3 + 4$, we have to subtract $(-4) \times$ column 3 from the right-hand side vector:

$$\begin{bmatrix} 9 \\ -3 \\ 15 \\ 10 \\ 8 \end{bmatrix} - (-4) \begin{bmatrix} 6 \\ 0 \\ 3 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 33 \\ -3 \\ 27 \\ 6 \\ 4 \end{bmatrix}$$

The upper bound of \bar{x}_3 becomes $\bar{u}_3 = -1 - (-4) = 3$, thus $0 \leq \bar{x}_3 \leq 3$ is obtained.

Additionally, there is a minus type variable x_2 that we want to convert into plus type. This can be achieved by changing the sign of all coefficients in column 2 and noting that now it represents $\bar{x}_2 = -x_2$.

1.2.1.3 Types of variables

Having performed the transformations of sections 1.2.1.1 and 1.2.1.2 we can assume that all MI variables have been reversed, all finite lower

bounds are zero and thus the problem is in form (1.11). We also assume that whatever transformations have been made are properly recorded so that when a solution is obtained with the transformed variables it can be expressed in terms of the original variables and constraints. For notational convenience, the j -th structural variable will be denoted by x_j regardless of the transformations it may have undergone. The same convention is followed for the other transformed items (lower and upper bounds, right-hand side elements and matrix coefficients).

Based on their feasibility range, variables (whether logical or structural) are categorized as follows:

Feasibility range	Type	Reference	Label
$z_i, x_j = 0$	0	Fixed	FX
$0 \leq z_i, x_j \leq u_j$	1	Bounded	BD
$0 \leq z_i, x_j \leq +\infty$	2	Nonnegative	PL
$-\infty \leq z_i, x_j \leq +\infty$	3	Free	FR

To refer to the type of a variable, notation $\text{type}(x_j)$ or $\text{type}(z_i)$ will be used.

There is a correspondence between the types of constraints and types of their logical variables in the general form of

$$z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = b_i, \quad i = 1, \dots, m,$$

namely,

$\text{type}(z_i)$	Constraint
0	Equality
1	Range
2	\geq or \leq
3	free (non-binding).

In the standard form of LP in section 1.1 all constraints are equalities and all variables are of type 2 (nonnegative).

1.2.1.4 Ultimate form #1

Let us assume that each general constraint has been converted into equality by adding an appropriate logical variable, all MI variables have been reversed, all finite lower bounds have been moved to zero, the resulting new variables are denoted by their corresponding original counterparts, new RHS values are also denoted by the original notation and

all changes have been recorded to enable the expression of the final solution in terms of the original variables and constraints. The LP problem is now:

$$\begin{aligned} \text{minimize} \quad & z = \sum_{j=1}^{\bar{n}} c_j x_j \\ \text{subject to} \quad & z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = b_i, \text{ for } i = 1, \dots, m \end{aligned}$$

and the type constraints on the variables.

Since c_0 of (1.7) does not play any role in any solution procedure it will be mostly ignored in the algorithms.

For practical reasons, the logical variables will be placed after the structurals. Then we have:

$$\min \quad \mathbf{c}^T \mathbf{x} \quad (1.15)$$

$$\text{s.t.} \quad \mathbf{Ax} + \mathbf{Iz} = \mathbf{b} \quad (1.16)$$

$$\text{and every variable is one of types 0--3.} \quad (1.17)$$

From a technical point of view, logical and structural variables play equal role in (1.14) – (1.16). In general, there is no need to distinguish them in theoretical investigations. We come back to this point at the details of implementation.

A more uniform representation can be achieved by the introduction of a simplified notation. We will use the assignment operator ‘:=’ to redefine an entity by using its old value or meaning in defining the new one. In this way, vectors \mathbf{x} and \mathbf{c} are redefined as

$$\mathbf{x} := \begin{bmatrix} \mathbf{x} \\ \mathbf{z} \end{bmatrix}, \quad \mathbf{c} := \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix}, \quad (1.17)$$

with $\mathbf{0}$ being the m dimensional null vector. The new dimension of \mathbf{x} and \mathbf{c} is denoted by n such that $n = \bar{n} + m$. A new upper bound vector \mathbf{u} is also defined

$$\mathbf{u} := \begin{bmatrix} \mathbf{u} \\ \mathbf{r} \end{bmatrix}, \quad (1.18)$$

where $\mathbf{r} \in \mathbb{R}^m$ (thus, the new $\mathbf{u} \in \mathbb{R}^n$) and

$$r_i = \begin{cases} 0 & \text{if type}(z_i) = 0, \\ R_i & \text{if type}(z_i) = 1, \\ +\infty & \text{otherwise.} \end{cases}$$

For completeness, an extended lower bound vector $\ell \in \mathbb{R}^n$ can also be defined with $\ell_j = -\infty$ if $\text{type}(x_j) = 3$ and $\ell_j = 0$ otherwise.

The matrix on the left hand side of (1.15) is redefined to be

$$\mathbf{A} := [\mathbf{A} \mid \mathbf{I}]. \quad (1.19)$$

With these notations, computational form #1 of the general LP problem, also called CF-1, can be rewritten as

$\min \quad \mathbf{c}^T \mathbf{x}$ s.t. $\mathbf{Ax} = \mathbf{b}$ (1.21) $\ell \leq \mathbf{x} \leq \mathbf{u}$.
--

In this form every x_j falls into one of the type categories $\{0, 1, 2, 3\}$.

The developments in this section have shown that every LP problem can be brought into this form. Furthermore, from the solution of (1.20), if it exists, the solution of the original problem of (1.7) – (1.9) can uniquely be determined.

Formally, (1.20) resembles the standard form of (1.1) – (1.3) except for the individual restrictions on the variables. However, the differences are more substantial. Matrix \mathbf{A} of (1.20) is a result of possible transformations and an augmentation. It contains a unit matrix, therefore it is of full row rank. Later, we will see that this form can be used in any version of the simplex method.

1.2.2 Computational form #2

While CF-1 in (1.20) is perfectly suitable for implementation there can be situations where the reduction of the types of variables and constraints makes the inclusion of new algorithmic techniques more complicated or renders them less efficient. To achieve a higher level of flexibility in this respect, it is possible to maintain more types. This section presents an alternative formulation that slightly differs from CF-1 and will be referred to as CF-2. The main difference is the introduction (or rather the keeping) of minus type variables and \geq type constraints.

1.2.2.1 Individual bounds

The cases of individual bounds on variables are, again, evaluated on the basis of the finiteness of ℓ_k and u_k in (1.9). As a result, PL, BD, FX and FR variables are defined exactly the same way as in CF-1. However, MI variables are not reversed but kept as they are defined by their individual bounds.

The general form of an MI variable is $-\infty \leq x_k \leq u_k < +\infty$. The finite upper bound can be translated to 0 by defining $\bar{x}_k = x_k - u_k$ from

which $x_k = \bar{x}_k + u_k$. Similarly to (1.12) and (1.13), we can write

$$\sum_{i=1}^m \mathbf{e}_i z_i + \sum_{j \neq k} \mathbf{a}_j x_j + \mathbf{a}_k (\bar{x}_k + u_k) = \mathbf{b},$$

which is further equal to

$$\sum_{i=1}^m \mathbf{e}_i z_i + \sum_{j \neq k} \mathbf{a}_j x_j + \mathbf{a}_k \bar{x}_k = \mathbf{b} - \mathbf{a}_k u_k. \quad (1.21)$$

It means column k of \mathbf{A} remains unchanged but now it represents \bar{x}_k . The RHS has to be transformed (updated) according to (1.21). When solution is obtained x_k can be restored from the above relationship.

1.2.2.2 General constraints

The discussion of special cases of constraints is very similar to that of section 1.2.1.2 but the \geq constraints will not be reversed. Therefore, if $U_i = +\infty$ and L_i is finite then (1.8) reduces to $\sum_{j=1}^{\bar{n}} a_j^i x_j \geq L_i$ (GE constraint). Using $b_i = L_i$ it can be converted into an equation

$$z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = b_i, \quad z_i \leq 0,$$

i.e., z_i is an MI variable.

All other types of constraints, namely, LE, RG, FX and FR are treated the same way as in CF-1.

As a result, we can conclude again that, with appropriately chosen z_i logical variables, the general constraints can be brought into the following form:

$$z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = b_i, \quad i = 1, \dots, m. \quad (1.22)$$

Using vector notation, (1.22) can be written as

$$\sum_{i=1}^m \mathbf{e}_i z_i + \sum_{j=1}^{\bar{n}} \mathbf{a}_j x_j = \mathbf{b},$$

or with matrix notation

$$\mathbf{Iz} + \mathbf{Ax} = \mathbf{b}.$$

It is important to remember that in this form neither the MI variables nor the GE constraints are reversed.

1.2.2.3 Types of variables

Now we have one more category of variables and constraints. Assuming that all finite lower or upper bounds have been translated to zero and the changes have been properly recorded for reconstruction, we have the following five types of variables (logical and structural alike):

Feasibility range	Type	Reference	Label
$z_i, x_j = 0$	0	Fixed	FX
$0 \leq z_i, x_j \leq u_j$	1	Bounded	BD
$0 \leq z_i, x_j \leq +\infty$	2	Nonnegative	PL
$-\infty \leq z_i, x_j \leq +\infty$	3	Free	FR
$-\infty \leq z_i, x_j \leq 0$	4	Nonpositive	MI

The correspondence between the types of constraints and types of their logical variables is also extended by one case. Remembering that all joint constraints are in the form of

$$z_i + \sum_{j=1}^n a_j^i x_j = b_i, \quad i = 1, \dots, m,$$

the relationship can be summarized in the following table:

type(z_i)	label(z_i)	Constraint	Label of constraint
0	FX	Equality	EQ
1	BD	Range	RG
2	PL	\leq	LE
3	FR	Non-binding	NB
4	MI	\geq	GE

It is instructive to overview the pros and cons of computational form #2.

Advantages:

- more flexibility,
- clear distinction between LE and GE constraints,
- less pre- and post-processing transformations.

Disadvantages:

- more types have to be considered,
- more administration is needed,
- some algorithmic procedures are more complicated.

1.2.2.4 Ultimate form #2

Let us assume that each general constraint has been converted into equality by adding an appropriate logical variable, all finite lower and MI type upper bounds have been moved to zero, the resulting new variables are denoted by their corresponding original counterparts, new RHS values are also denoted by the original notation and all changes have been recorded to enable the expression of the solution in terms of the original variables and constraints. The LP problem is now:

$$\begin{aligned} \min \quad & z = \sum_{j=1}^{\bar{n}} c_j x_j \\ \text{s.t.} \quad & z_i + \sum_{j=1}^{\bar{n}} a_j^i x_j = b_i, \text{ for } i = 1, \dots, m, \end{aligned}$$

and the type constraints on the variables.

Reversing the order of logical and structural variables and using matrix notation:

$$\min \quad \mathbf{c}^T \mathbf{x} \tag{1.24}$$

$$\text{s.t.} \quad \mathbf{A}\mathbf{x} + \mathbf{I}\mathbf{z} = \mathbf{b} \tag{1.25}$$

$$\text{and every variable is one of types 0–4.} \tag{1.26}$$

Logical and structural variables can be considered in a uniform way. As a result, a simplified notation can, again, be introduced by denoting $n := m + \bar{n}$ and redefining vectors \mathbf{x} and \mathbf{c} as in (1.17). The definition of the new \mathbf{u} , however, differs slightly from (1.18) to accommodate the (logical or structural) MI variables. Therefore, in

$$\mathbf{u} := \begin{bmatrix} \mathbf{u} \\ \mathbf{r} \end{bmatrix}$$

\mathbf{r} is m dimensional and

$$r_i = \begin{cases} 0 & \text{if } \text{type}(z_i) = 0, \text{ or } \text{type}(z_i) = 4 \\ R_i & \text{if } \text{type}(z_i) = 1, \\ +\infty & \text{otherwise.} \end{cases}$$

The extended $\ell \in \mathbb{R}^n$ lower bound vector is now defined as

$$\ell_j = \begin{cases} -\infty & \text{if } \text{type}(x_j) = 3, \text{ or } \text{type}(x_j) = 4 \\ 0 & \text{otherwise.} \end{cases}$$

The above definitions enable the introduction of a simplified notation similar to (1.19) to obtain computational form #2 (CF-2) of the general LP

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{Ax} = \mathbf{b} \quad (1.27) \\ & \ell \leq \mathbf{x} \leq \mathbf{u}. \end{aligned}$$

In this form, $\text{type}(x_j) \in \{0, 1, 2, 3, 4\}, j = 1, \dots, n$.

Comments at the end of computational form #1 equally apply to CF-2 and are not repeated here.

1.3. Concluding remarks

1.3.1 Translation of bounds

Translating all finite lower bounds and the upper bounds of type-4 structural variables to 0 has practical advantages. First, comparisons like $x_j \geq \ell_j$ can be replaced by a simple sign test on x_j . Similarly, if a tolerance of ϵ is allowed then the tests are $x_j \geq \ell_j - \epsilon$ and $x_j \geq \epsilon$, respectively, again, the latter being simpler (and faster to execute). Second, the finite nonzero lower and upper bounds are not explicitly used during computations. As a consequence, they need not be stored in the computer's memory.

However, depending on the algorithmic techniques, the explicit use of lower and upper bounds can increase the algorithmic richness of a simplex solver. If the creation of a flexible system is the goal then it is certainly a good idea to include the lower and upper bounds in the data structure and also in the algorithms. Whether or not it is done, the two computational forms discussed in sections 1.2.1 and 1.2.2 still can be used with obvious minor adjustments.

1.3.2 Yet another formulation

There is one more interesting version that deserves mentioning. If we define

$$y_i = \sum_{j=1}^{\bar{n}} a_j^i x_j \quad (1.27)$$

then constraint (1.8) can be written as

$$L_i \leq y_i \leq U_i.$$

The advantage of this notation is that the general form (1.7) – (1.9) of the LP problem can now be written in the bit more unified form

$$\begin{aligned} \text{minimize} \quad & z = c_1 x_1 + \cdots + c_{\bar{n}} x_{\bar{n}} \\ \text{subject to} \quad & y_i = \sum_{j=1}^{\bar{n}} a_j^i x_j, \quad i = 1, \dots, m, \end{aligned} \quad (1.28)$$

$$L_i \leq y_i \leq U_i, \quad i = 1, \dots, m, \quad (1.29)$$

$$\ell_j \leq x_j \leq u_j, \quad j = 1, \dots, \bar{n}. \quad (1.30)$$

The difference between this formulation and the two computational forms lies in the definition of the logical variables associated with the constraints. In sections 1.2.1.2 and 1.2.2.2 z_i is defined as the difference between the left and right-hand sides, $z_i = b_i - \sum_{j=1}^{\bar{n}} a_j^i x_j$, while now y_i is defined to be the value of the left hand side, see (1.27).

It may seem that this form has more constraints and thus defines a larger problem than computational forms #1 and #2. However, the newly added constraints in (1.29) are just individual bounds that correspond to the constraints on the logical variables which are also present in (1.20) and (1.26). Therefore, formulation (1.28) – (1.30) is mathematically equivalent to both computational forms discussed. There are cases when this new form offers some advantages due to its uniform treatment of bounds on variables and constraints.

Chapter 2

THE SIMPLEX METHOD

The main solution algorithm for the LP problem has long been the *simplex method* introduced by Dantzig. Since its inception, the algorithm has gone through a substantial evolution but its basic principles have remained the same. The simplex method is a computationally demanding procedure that requires the use of computers. It is widely accepted that the growing need for the solution of real-life LP problems has significantly inspired the development of computers in the 1950s and early 1960s [Dantzig, 1988].

From theoretical point of view the simplex method is not necessarily a promising algorithm as its worst case performance is exponential. It means that the number of iterations needed to solve a problem is bounded by an exponential function of m and n . Examples have been created for which the iteration count of the algorithm reaches its theoretical maximum [Klee and Minty, 1972]. In practice, however, the simplex method shows an average performance which is a linear function of m and it is highly efficient in solving real-life problems.

Between 1951 and the mid 1970's the main aim of the researchers was to enhance the computational capabilities of the simplex method. By the end of this period it was believed that the simplex method had reached its maturity and was not worth much further effort. However, in the mid 1980's the introduction of interior point methods (IPMs) for solving the LP problem revived the interest in the simplex. To the surprise of many observers the simplex method has made tremendous progress during the rivalry with the IPMs. The gain in efficiency is estimated to be as much as two orders of magnitude. This development is due to several factors. One of the main purposes of this book is to give account of the details of algorithms and issues of implementation that have lead to this success.

This chapter presents the theoretical background and the basic algorithms of the simplex method. The main aim of this discussion is to establish the foundations of the subsequent advanced topics.

2.1. Theoretical background

In this chapter we consider computational form #1 with the simplifying assumption that all variables are of type 2.

$$(P1) \quad \text{minimize} \quad z = \mathbf{c}^T \mathbf{x} \quad (2.1)$$

$$\text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b} \quad (2.2)$$

$$\mathbf{x} \geq \mathbf{0}, \quad (2.3)$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and the vectors are of compatible dimensions. Since \mathbf{A} contains an $m \times m$ identity matrix for the logical variables it is of full row rank and $m < n$.

Any vector \mathbf{x} that satisfies (2.2) is called a *solution*. If a solution satisfies the (2.3) nonnegativity restrictions it is called a *feasible solution*. The set of feasible solutions is denoted by \mathcal{X} , i.e.,

$$\mathcal{X} = \{\mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}. \quad (2.4)$$

If $\mathcal{X} = \emptyset$ we say the problem is infeasible.

PROPOSITION 2.1.1 *If \mathcal{X} is nonempty it is a convex set.*

PROOF. Let \mathbf{x} and \mathbf{y} be two feasible solutions, i.e., $\mathbf{x}, \mathbf{y} \geq \mathbf{0}$ and $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A}\mathbf{y} = \mathbf{b}$. We show that the convex linear combination of \mathbf{x} and \mathbf{y} is also a feasible solution. Obviously, if $0 \leq \lambda \leq 1$ then $\lambda\mathbf{x} + (1 - \lambda)\mathbf{y} \geq \mathbf{0}$. Furthermore, $\mathbf{A}(\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}) = \lambda\mathbf{A}\mathbf{x} + (1 - \lambda)\mathbf{A}\mathbf{y} = \lambda\mathbf{b} + (1 - \lambda)\mathbf{b} = \mathbf{b}$, which proves the proposition. \square

A feasible solution \mathbf{x}^* is called *optimal* if $\mathbf{c}^T \mathbf{x}^* \leq \mathbf{c}^T \mathbf{x}$ for all $\mathbf{x} \in \mathcal{X}$. There may be many optimal solutions with the same objective value. If $\mathbf{x}_1, \dots, \mathbf{x}_k$ are all optimal solutions with $\mathbf{c}^T \mathbf{x}_1 = \dots = \mathbf{c}^T \mathbf{x}_k = z^*$ then their convex linear combination $\lambda_1 \mathbf{x}_1 + \dots + \lambda_k \mathbf{x}_k$, where $\lambda_i \geq 0 \forall i$ and $\sum \lambda_i = 1$, is also an optimal solution, because

$$\begin{aligned} \mathbf{c}^T(\lambda_1 \mathbf{x}_1 + \dots + \lambda_k \mathbf{x}_k) &= \lambda_1 \mathbf{c}^T \mathbf{x}_1 + \dots + \lambda_k \mathbf{c}^T \mathbf{x}_k \\ &= (\lambda_1 + \dots + \lambda_k) z^* \\ &= z^*. \end{aligned} \quad (2.5)$$

If the optimum is not unique in the above sense then we say the problem has *alternative optima*.

2.1.1 Basis and solution

Let the index set of all variables be denoted by $\mathcal{N} = \{1, \dots, n\}$. The number of variables is $|\mathcal{N}| = n$.

In (2.2), every variable x_j has a column vector \mathbf{a}_j associated with it. Select m linearly independent columns $\mathbf{a}_{k_1}, \dots, \mathbf{a}_{k_m}$ from \mathbf{A} . They form a *basis* of \mathbb{R}^m and the corresponding variables are called *basic variables*. The ordered index set of these variables is denoted by \mathcal{B} , i.e.,

$$\mathcal{B} = \{k_1, \dots, k_m\}. \quad (2.6)$$

The index set of the ‘remaining’ variables (which are nonbasic) is denoted by \mathcal{R} . If we use \mathcal{B} as a subscript to a matrix like $\mathbf{A}_{\mathcal{B}}$ we think of the submatrix of \mathbf{A} consisting of all rows of \mathbf{A} and the columns listed in \mathcal{B} . Subvectors $\mathbf{c}_{\mathcal{B}}$ and $\mathbf{x}_{\mathcal{B}}$ are defined similarly. For example, if \mathbf{A} has 3 rows and 7 columns then we can have $\mathcal{B} = \{2, 6, 5\}$ and $\mathcal{R} = \{1, 3, 4, 7\}$. The order of the variables in \mathcal{B} is meaningful.

It can be assumed, without loss of generality, that columns belonging to \mathcal{B} are permuted to the first m positions of \mathbf{A} . In this way, \mathbf{A} can be written in a partitioned form as $\mathbf{A} = [\mathbf{A}_{\mathcal{B}} | \mathbf{A}_{\mathcal{R}}]$. The similar partitioning of \mathbf{c} and \mathbf{x} is $\mathbf{c} = \begin{bmatrix} \mathbf{c}_{\mathcal{B}} \\ \mathbf{c}_{\mathcal{R}} \end{bmatrix}$ and $\mathbf{x} = \begin{bmatrix} \mathbf{x}_{\mathcal{B}} \\ \mathbf{x}_{\mathcal{R}} \end{bmatrix}$. Now $\mathcal{B} = \{1, \dots, m\}$. For notational convenience, we write $\mathbf{B} = \mathbf{A}_{\mathcal{B}}$, $\mathbf{R} = \mathbf{A}_{\mathcal{R}}$ and

$$\mathbf{A} = [\mathbf{B} | \mathbf{R}]. \quad (2.7)$$

In our example, $\mathbf{B} = [\mathbf{a}_2, \mathbf{a}_6, \mathbf{a}_5]$, $\mathbf{c}_{\mathcal{B}} = [c_2, c_6, c_5]^T$ and $\mathbf{x}_{\mathcal{B}} = [x_2, x_6, x_5]^T$ while $\mathbf{R} = \{\mathbf{a}_1, \mathbf{a}_3, \mathbf{a}_4, \mathbf{a}_7\}$, $\mathbf{c}_{\mathcal{R}} = [c_1, c_3, c_4, c_7]^T$ and $\mathbf{x}_{\mathcal{R}} = [x_1, x_3, x_4, x_7]^T$.

Obviously, $\mathcal{N} = \mathcal{B} \cup \mathcal{R}$ and $\mathcal{B} \cap \mathcal{R} = \emptyset$. Furthermore, $|\mathcal{B}| = m$ and $|\mathcal{R}| = n - m$.

Substituting (2.7) into (2.2) we obtain

$$\mathbf{B}\mathbf{x}_{\mathcal{B}} + \mathbf{R}\mathbf{x}_{\mathcal{R}} = \mathbf{b}. \quad (2.8)$$

As \mathbf{B} is nonsingular (its columns are linearly independent) its inverse, \mathbf{B}^{-1} , exists. Multiplying both sides of (2.8) by \mathbf{B}^{-1} and rearranging the equation we can express $\mathbf{x}_{\mathcal{B}}$ as

$$\mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_{\mathcal{R}}). \quad (2.9)$$

This equation shows that the nonbasic variables uniquely determine the values of the basic variables. Therefore, we can think of basic variables as dependent and the nonbasic variables as independent variables. If all nonbasic variables are zero, (2.9) becomes

$$\mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}\mathbf{b}. \quad (2.10)$$

A solution with $\mathbf{x}_{\mathcal{R}} = \mathbf{0}$ is called a *basic solution*. If a basic solution (2.10) satisfies the (2.3) nonnegativity restrictions it is called a *basic feasible solution*. A basic feasible solution is often referred to by the acronym BFS. With a given \mathcal{B} , a BFS is characterized by $\mathbf{x}_{\mathcal{B}} \geq \mathbf{0}$ and $\mathbf{x}_{\mathcal{R}} = \mathbf{0}$. We say that a basis is feasible if the corresponding BFS is feasible.

If $\mathbf{x}_{\mathcal{B}} > \mathbf{0}$ it is called a *nondegenerate BFS*, otherwise, i.e., when at least one component of $\mathbf{x}_{\mathcal{B}}$ is zero, it is a *degenerate BFS*. We also say the corresponding basis \mathcal{B} is nondegenerate or degenerate, respectively. The *degree of degeneracy* is the number of components in $\mathbf{x}_{\mathcal{B}}$ at zero level.

The following theorem is of fundamental importance in linear programming.

THEOREM 2.1 *If the system of constraints (2.2) – (2.3) has a feasible solution, i.e., $\mathcal{X} = \{\mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\} \neq \emptyset$, then it also has a basic feasible solution.*

PROOF. Let $\mathbf{x} \in \mathcal{X}$ and $\mathbf{x} \neq \mathbf{0}$. (For $\mathbf{x} = \mathbf{0}$ the statement is trivially true because of the presence of a unit matrix in \mathbf{A} .) We can assume that the nonzero components of \mathbf{x} are located in the first positions such that $x_j > 0$, for $1 \leq j \leq k$ and $x_j = 0$ for $k + 1 \leq j \leq n$.

Let vectors $\mathbf{a}_1, \dots, \mathbf{a}_k$ be linearly independent. If $k = m$, we are done. If $k < m$ we can add $m - k$ vectors from $\mathbf{a}_{k+1}, \dots, \mathbf{a}_n$ to make a full basis. Such vectors exist because $\text{rank}(\mathbf{A}) = m$ and the statement is true again.

Let us now assume that vectors $\mathbf{a}_1, \dots, \mathbf{a}_k$ are linearly dependent. In this case there are real numbers $\lambda_1, \dots, \lambda_k$ with at least one of them different from zero such that

$$\lambda_1 \mathbf{a}_1 + \cdots + \lambda_k \mathbf{a}_k = \mathbf{0}. \quad (2.11)$$

If all nonzero λ 's are negative we multiply (2.11) by -1 to make at least one λ positive. Due to the assumptions, the joint constraints can be written as

$$x_1 \mathbf{a}_1 + \cdots + x_k \mathbf{a}_k = \mathbf{b}. \quad (2.12)$$

(2.11) can be used to eliminate at least one vector from the expression of \mathbf{b} in such a way that the multipliers of \mathbf{a}_j 's remain nonnegative in (2.12). Choose subscript q to satisfy

$$\frac{x_q}{\lambda_q} = \min_{\lambda_j > 0} \frac{x_j}{\lambda_j}. \quad (2.13)$$

Multiplying (2.11) by $\frac{x_q}{\lambda_q}$ and subtracting it from (2.12), we obtain

$$\sum_{j=1}^k \left(x_j - \frac{x_q}{\lambda_q} \lambda_j \right) \mathbf{a}_j = \mathbf{b}. \quad (2.14)$$

By (2.13), the multiplier of each \mathbf{a}_j in (2.14) is nonnegative and the multiplier of \mathbf{a}_q is 0. Thus, we have obtained a feasible solution with no more than $k - 1$ positive components. If the vectors in (2.14) with positive components are linearly independent, the procedure terminates. If not, we repeat it. In a finite number of steps a solution is reached which has no more than m linearly independent vectors and their multipliers are nonnegative. If there are less than m vectors left we can add appropriate vectors from $\mathbf{a}_{k+1}, \dots, \mathbf{a}_n$, as above, to make a basis for \mathbb{R}^m . \square

2.1.2 The geometry of constraints

Feasibility set \mathcal{X} defined in (2.4) is a bounded or unbounded convex polyhedral set. In \mathbb{R}^2 such sets look like the shaded areas in Figures 2.1 and 2.2, respectively. In general, \mathcal{X} is bounded if there exists a number $M > 0$ such that $\|\mathbf{x}\|_2 \leq M$ for all $\mathbf{x} \in \mathcal{X}$, where $\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^T \mathbf{x}}$ (the Euclidean norm of \mathbf{x}).

Geometrically, the basic feasible solutions correspond to the vertices of \mathcal{X} , like A, B, C, D and E in Figure 2.1. The vertices are the extreme points of \mathcal{X} as they cannot be written as a nontrivial linear combination of the points in \mathcal{X} . They lie at the intersection of n hyperplanes that define \mathcal{X} . Note, the nonnegativity constraints also define hyperplanes, so in (2.4) we have $m + n$ of them. These facts are well investigated in the theory of convex sets.

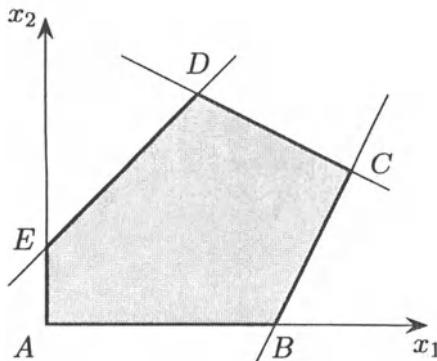


Figure 2.1. A two dimensional convex polyhedron with five extreme points (vertices).

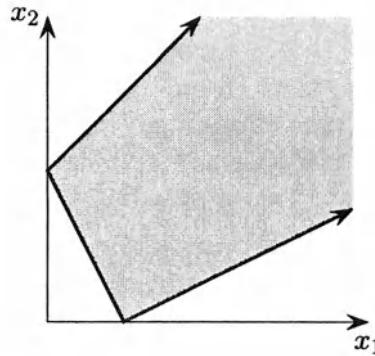


Figure 2.2. A two dimensional unbounded convex polyhedral set with two extreme points (vertices) and two extreme directions (lines with arrows).

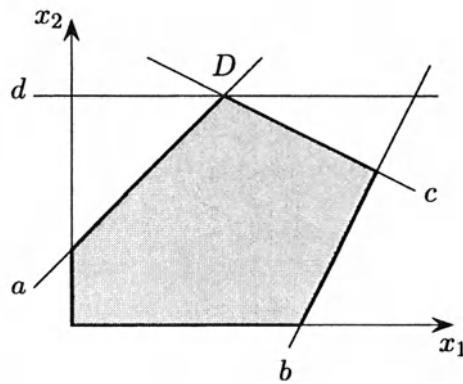


Figure 2.3. A two dimensional convex polyhedron with degeneracy at vertex D .

We say that a vertex is *degenerate* if more than n hyperplanes intersect in it. For instance, vertex D in Figure 2.3 can be viewed as the intersection of lines (two dimensional hyperplanes) a and c or a and d or, finally, c and d . In other words, there is ambiguity in the representation of a degenerate vertex. For the simplex method, such vertices require special attention as they may cause algorithmic difficulties, as will be pointed out later.

In case \mathcal{X} is unbounded then, in addition to the extreme points, we have extreme directions. They are illustrated by the arrows in Figure 2.2. The following theorem, which is presented without proof, characterizes the special importance of extreme points and directions.

THEOREM 2.2 (REPRESENTATION THEOREM) *If \mathcal{X} is a nonempty set then the set of vertices is finite, say $\mathbf{x}_1, \dots, \mathbf{x}_k$, and the set of extreme directions is also finite, say $\mathbf{y}_1, \dots, \mathbf{y}_q$. Furthermore, any point $\mathbf{x} \in \mathcal{X}$ can be represented as*

$$\mathbf{x} = \sum_{i=1}^k \lambda_i \mathbf{x}_i + \sum_{j=1}^q \mu_j \mathbf{y}_j, \quad \sum_{i=1}^k \lambda_i = 1, \quad \lambda_i \geq 0, \quad \forall i \text{ and } \mu_j \geq 0, \quad \forall j.$$

With this theorem we can characterize a very important feature of linear programming which is fundamental for the simplex method.

THEOREM 2.3 *Let $\mathcal{X} \neq \emptyset$, i.e., \mathcal{X} of (2.4) is nonempty. Then either (a) the solution of LP problem (P1) is unbounded or (b) (P1) has an optimal basic feasible solution.*

PROOF. Let \mathbf{x} be a feasible solution to (P1). Due to the representation theorem, the corresponding objective value can be written as

$$z = \mathbf{c}^T \mathbf{x} = \sum_{i=1}^k \lambda_i \mathbf{c}^T \mathbf{x}_i + \sum_{j=1}^q \mu_j \mathbf{c}^T \mathbf{y}_j, \quad (2.15)$$

with $\sum_{i=1}^k \lambda_i = 1, \lambda_i \geq 0, \forall i$. In the second sum, if $\mathbf{c}^T \mathbf{y}_j < 0$ holds for any j , z can be decreased infinitely by making μ_j arbitrarily large. This proves assertion (a).

Assume that the problem has a finite optimum attained at \mathbf{x}^* with a value of $z^* = \mathbf{c}^T \mathbf{x}^*$. Now $\mathbf{c}^T \mathbf{y}_j \geq 0$ holds for all j otherwise it is case (a). We show that if \mathbf{x}^* is not a vertex then there exist a vertex where the objective value is equal to z^* . Assume the opposite, i.e., at every extreme point the objective is worse than at \mathbf{x}^* , i.e.,

$$\mathbf{c}^T \mathbf{x}^* < \mathbf{c}^T \mathbf{x}_i, \quad i = 1, \dots, k. \quad (2.16)$$

Dropping the second sum (which is nonnegative) from the right-hand side of (2.15) and using (2.16) we obtain

$$\mathbf{c}^T \mathbf{x}^* \geq \sum_{i=1}^k \lambda_i \mathbf{c}^T \mathbf{x}_i > \sum_{i=1}^k \lambda_i \mathbf{c}^T \mathbf{x}^* = \mathbf{c}^T \mathbf{x}^* \sum_{i=1}^k \lambda_i = \mathbf{c}^T \mathbf{x}^*,$$

which is a contradiction. Therefore, there must be at least one index i such that $\mathbf{c}^T \mathbf{x}^* = \mathbf{c}^T \mathbf{x}_i$, which proves (b). \square

The message of this theorem is that it is sufficient to investigate the basic feasible solutions. If there is a finite optimum there is also a vertex where the optimum is attained.

The number of theoretically possible bases is $\binom{n}{m}$ which is the number of times m different columns can be taken out of n . Even for moderate values of m and n this expression can be astronomically large. As it will be shown, the simplex method vastly reduces the investigation of the number of vertices by making an intelligent search of all bases.

2.1.3 Neighboring bases

Two bases of \mathbb{R}^m formed by the columns of \mathbf{A} are called *neighboring* if they differ from each other only in one column. More formally, let columns of \mathbf{B} be denoted by \mathbf{b}_i , i.e., $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_{p-1}, \mathbf{b}_p, \mathbf{b}_{p+1}, \dots, \mathbf{b}_m]$, where each \mathbf{b}_i is a column of \mathbf{A} so that $\mathbf{b}_i = \mathbf{a}_{k_i}$. Let the neighboring basis be defined by the same set of columns except the p -th which is replaced by another column of \mathbf{A} . For notational simplicity, this vector is denoted by \mathbf{a} . The neighboring basis is thus $\mathbf{B}_a = [\mathbf{b}_1, \dots, \mathbf{b}_{p-1}, \mathbf{a}, \mathbf{b}_{p+1}, \dots, \mathbf{b}_m]$.

Since \mathbf{a} is an m dimensional vector it can be written as a linear combination of the columns of \mathbf{B} :

$$\mathbf{a} = \sum_{i=1}^m v_i \mathbf{b}_i, \quad (2.17)$$

which can further be written as $\mathbf{a} = \mathbf{B}\mathbf{v}$ if notation $\mathbf{v} = [v_1, \dots, v_m]^T$ is used. Since \mathbf{B} is nonsingular,

$$\mathbf{v} = \mathbf{B}^{-1} \mathbf{a}. \quad (2.18)$$

If $v_p \neq 0$, \mathbf{b}_p can be expressed from (2.17) as

$$\mathbf{b}_p = \frac{1}{v_p} \mathbf{a} - \sum_{i \neq p} \frac{v_i}{v_p} \mathbf{b}_i \quad (2.19)$$

Vectors on the RHS of (2.19) form \mathbf{B}_a . It is known from elementary linear algebra that \mathbf{B}_a is nonsingular if and only if $v_p \neq 0$.

(2.19) is nothing but the expression of \mathbf{b}_p in terms of \mathbf{B}_a . It can be written as $\mathbf{b}_p = \mathbf{B}_a \boldsymbol{\eta}$, where

$$\boldsymbol{\eta} = \left[-\frac{v_1}{v_p}, \dots, -\frac{v_{p-1}}{v_p}, \frac{1}{v_p}, -\frac{v_{p+1}}{v_p}, \dots, -\frac{v_m}{v_p} \right]^T.$$

For future reference, it is useful to remember the definition of the components of $\boldsymbol{\eta}$:

$$\eta^p = \frac{1}{v_p} \quad (2.20)$$

$$\eta^i = -\frac{v_i}{v_p} \quad (\text{or } \eta^i = -v_i \eta^p) \quad \text{for } i \neq p, \quad (2.21)$$

where p is often referred to as *pivot position* and v_p as *pivot element*.

If we define matrix \mathbf{E} as

$$\mathbf{E} = [\mathbf{e}_1, \dots, \mathbf{e}_{p-1}, \boldsymbol{\eta}, \mathbf{e}_{p+1}, \dots, \mathbf{e}_m] \quad (2.22)$$

then all the original basis vectors can be expressed in terms of the new basis as

$$\mathbf{B} = \mathbf{B}_a \mathbf{E}. \quad (2.23)$$

From (2.22), we can see that \mathbf{E} has a very simple structure. It differs from the unit matrix in just one column. This column is often referred to as $\boldsymbol{\eta}$ (eta) vector while the matrix itself is called Elementary Transformation Matrix (ETM):

$$\mathbf{E} = \begin{bmatrix} 1 & \eta^1 & & \\ \ddots & \vdots & & \\ & \eta^p & & \\ & \vdots & \ddots & \\ & \eta^m & & 1 \end{bmatrix} \quad (2.24)$$

To fully represent \mathbf{E} only the eta vector and its position in the unit matrix need to be recorded.

In each iteration, the simplex method moves from a basis to a neighboring one until termination. The operations of an iteration are defined in terms of the inverse of the current basis. Determining \mathbf{B}^{-1} is a substantial computational effort. If it had to be computed in every iteration it would be computationally prohibitive. The development of this section makes it possible to obtain \mathbf{B}_a^{-1} easily if \mathbf{B}^{-1} is known.

Taking the inverse of both sides of (2.23) we get $\mathbf{B}^{-1} = \mathbf{E}^{-1} \mathbf{B}_a^{-1}$. If this equation is multiplied by \mathbf{E} the result is

$$\boxed{\mathbf{B}_a^{-1} = \mathbf{E} \mathbf{B}^{-1}}. \quad (2.25)$$

Verbally: the inverse of a neighboring basis can be obtained from the original inverse by premultiplying it by an ETM which is derived from the basis change. Namely, the incoming vector \mathbf{a} determines $\mathbf{v} = \mathbf{B}^{-1} \mathbf{a}$ needed for $\boldsymbol{\eta}$ and the outgoing vector (more precisely, its position p in the basis) determines $\boldsymbol{\eta}$.

Operation defined in (2.25) is called *updating* \mathbf{B}^{-1} . If the rows of \mathbf{B}^{-1} are denoted by $\boldsymbol{\rho}^i$, $i = 1, \dots, m$, then the effect of updating \mathbf{B}^{-1} is the following. Compute the new row p as $\bar{\boldsymbol{\rho}}^p = (1/v_p)\boldsymbol{\rho}^p$ and determine the other new rows from $\bar{\boldsymbol{\rho}}^i = \boldsymbol{\rho}^i - v_i \bar{\boldsymbol{\rho}}^p$ for $i = 1, \dots, m$, $i \neq p$. \mathbf{B}_a^{-1} consists of rows $\bar{\boldsymbol{\rho}}^1, \dots, \bar{\boldsymbol{\rho}}^m$.

2.2. The primal simplex method

The main idea of the primal simplex method is the following. If a problem has an optimal solution then there is a basic solution which is also optimal. Therefore, it is enough to deal with basic solutions. The simplex method starts with a basic feasible solution. If it is not optimal then a sequence of neighboring feasible bases is determined with monotonically improving objective value until an optimal solution is reached or the unboundedness of the solution is detected.

The primal simplex method is due to Dantzig. The derivation below is somewhat different from the version found in his book [Dantzig, 1963] but it serves later purposes better. In compact form, the problem to solve is $\min\{\mathbf{c}^T \mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$.

2.2.1 Optimality conditions

Let us assume \mathcal{X} of (2.4) in nonempty and a feasible basis \mathcal{B} is known. According to (2.9), $\mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_{\mathcal{R}})$. Substituting this $\mathbf{x}_{\mathcal{B}}$ into the partitioned form of the objective function, $z = \mathbf{c}_{\mathcal{B}}^T \mathbf{x}_{\mathcal{B}} + \mathbf{c}_{\mathcal{R}}^T \mathbf{x}_{\mathcal{R}}$, we obtain $\mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_{\mathcal{R}}) + \mathbf{c}_{\mathcal{R}}^T \mathbf{x}_{\mathcal{R}}$ which can be rearranged as

$$z = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{b} + (\mathbf{c}_{\mathcal{R}}^T - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{R}) \mathbf{x}_{\mathcal{R}}. \quad (2.26)$$

Now the objective value is expressed as a function of the current nonbasic variables. Component j of $\mathbf{c}_{\mathcal{R}}^T - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{R}$ (the multiplier of x_j in $\mathbf{x}_{\mathcal{R}}$) is called the *reduced cost (RC)*, or updated objective coefficient, of x_j and is denoted by d_j :

$$d_j = c_j - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{a}_j. \quad (2.27)$$

Vector

$$\boldsymbol{\pi}^T = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \quad (2.28)$$

is of special importance and is called *simplex multiplier*. One of its use is to determine the reduced cost as

$$d_j = c_j - \boldsymbol{\pi}^T \mathbf{a}_j. \quad (2.29)$$

PROPOSITION 2.2.1 *The reduced cost of basic variables is equal to zero.*

PROOF. Applying the definition of the reduced cost to the i -th basic variable x_{k_i} , we obtain $d_{k_i} = c_{k_i} - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{a}_{k_i}$. As x_{k_i} is in position i of the basis, $\mathbf{B}^{-1} \mathbf{a}_{k_i} = \mathbf{e}_i$. The i -th component of $\mathbf{c}_{\mathcal{B}}$ is c_{i_k} . Thus, $d_{k_i} = c_{k_i} - \mathbf{c}_{\mathcal{B}}^T \mathbf{e}_i = c_{k_i} - c_{k_i} = 0$. \square

As \mathbf{B} is a feasible basis, we can check whether it is optimal. From (2.9) we know that the values of the basic variables are uniquely determined

by the nonbasic variables. We can try to give a positive value to a nonbasic variable, say x_j , $j \in \mathcal{R}$ and observe what happens. From (2.26) and (2.27), it can improve (decrease) the objective function if $d_j < 0$. As the result of this attempt, the values of the basic variables will change, according to (2.9). Since we have to maintain the feasibility of the solution, it may not be possible to find such a positive value for x_j . On the other hand, if $d_j > 0$ then changing the value of x_j from its currently feasible level of 0 the objective function would deteriorate (increase). Furthermore, if $d_j = 0$ then changing the value of x_j does not affect the value of the objective function. Therefore, we can conclude that if $d_j \geq 0$ for all $j \in \mathcal{R}$ there is no feasible solution $\mathbf{x}' \in \mathcal{X}$ such that $\mathbf{c}^T \mathbf{x}' < \mathbf{c}^T \mathbf{x}$. Because of the convexity of \mathcal{X} and the objective function, it means that \mathbf{x} is an optimal solution. Thus, a sufficient condition for a feasible basis \mathcal{B} to be optimal, also referred to as the *optimality condition*, is that

$$d_j \geq 0, \text{ for all } j \in \mathcal{R}. \quad (2.30)$$

It is important to remember that (2.30) is a sufficient condition. There can be optimal bases that do not satisfy this condition. An example (due to Majthay, also shown by Prékopa [Prékopa, 1995]) demonstrates such a situation:

$$\begin{aligned} \min \quad & x_1 + 2x_2 \\ \text{s.t.} \quad & x_1 + x_2 = 0, \\ & x_1, x_2 \geq 0. \end{aligned}$$

While the only solution to this problem is $x_1 = x_2 = 0$, there are two bases and both are optimal. One of them is the 1×1 matrix representing $\mathcal{B} = \{1\}$. With $a_1^1 = 1$ which gives $d_2 = 1$. The other is $\mathcal{B} = \{2\}$ with $a_1^2 = 1$ yielding $d_1 = -1$ which does not satisfy (2.30).

If the objective function in (2.1) is to be maximized the sign of d_j in the optimality condition is the opposite, i.e., $d_j \leq 0$, for all $j \in \mathcal{R}$.

2.2.2 Improving a nonoptimal basic feasible solution

The simplex method tries to find a basic feasible solution that satisfies the (2.30) sufficient conditions of optimality. The idea is that if a feasible basis does not satisfy these conditions then a neighboring feasible basis can be found with a better or not worse objective value.

Let us assume, the reduced cost of x_q violates the optimality condition, i.e., $d_q < 0$. We can try to move it away from 0 in such a way that

- (i) the main equations in (2.9) are maintained,
- (ii) all basic variables remain feasible, and
- (iii) the objective function improves as much as possible.

Let us denote the displacement of x_q by t . As all other nonbasic variables are kept at zero, the main equations, as a function of t take the form of

$$\mathbf{Bx}_B(t) + t\mathbf{a}_q = \mathbf{b},$$

from which

$$\mathbf{x}_B(t) = \mathbf{B}^{-1}\mathbf{b} - t\mathbf{B}^{-1}\mathbf{a}_q. \quad (2.31)$$

Introducing notation

$$\boldsymbol{\alpha}_q = \mathbf{B}^{-1}\mathbf{a}_q \quad (2.32)$$

$$\boldsymbol{\beta} = \mathbf{B}^{-1}\mathbf{b} \quad (2.33)$$

$$\boldsymbol{\beta}(t) = \mathbf{x}_B(t) \quad (2.34)$$

(2.31) can be rewritten as

$$\boldsymbol{\beta}(t) = \boldsymbol{\beta} - t\boldsymbol{\alpha}_q. \quad (2.35)$$

$\boldsymbol{\alpha}_q$ is also referred to as *search direction* and t as *steplength*. For $t = 0$, we obtain the current basic feasible solution $\boldsymbol{\beta}(0) = \boldsymbol{\beta} = \mathbf{x}_B$. The i -th component of (2.35) is

$$\beta_i(t) = \beta_i - t\alpha_q^i \quad (2.36)$$

(2.36) enables us to determine the steplength so that all three requirements are satisfied. The feasibility of the basic variables requires that for $t \geq 0$

$$\beta_i(t) = \beta_i - t\alpha_q^i \geq 0, \quad \text{for } i = 1, \dots, m, \quad (2.37)$$

is maintained. Depending on the sign of α_q^i , we can distinguish two cases. If $\alpha_q^i > 0$ then $t \leq \beta_i/\alpha_q^i$. Let

$$\mathcal{I} = \{i : \alpha_q^i > 0\}. \quad (2.38)$$

It is easy to see that the largest value of t for which (2.37) still holds is determined by

$$\theta = \frac{\beta_p}{\alpha_q^p} = \min_{i \in \mathcal{I}} \left\{ \frac{\beta_i}{\alpha_q^i} \right\}. \quad (2.39)$$

This operation is called *ratio test*. We say that the p -th basic variable is the *blocking variable* because any further increase of t would drive this variable negative (infeasible). α_q^p is called the pivot element. The minimum may not be unique. In this case any position with the minimum

ratio can be taken as pivot. This freedom gives a flexibility which is very useful from computational point of view. On the other hand, it leads to degeneracy which is a less fortunate consequence. More details on this issue are coming later.

If $\alpha_q^i \leq 0$ then (2.37) is true for any $t \geq 0$, therefore, such positions do not impose any restriction on the maximum of t . If $\alpha_q^i \leq 0$ for all i then $\mathcal{I} = \emptyset$ and the displacement of x_q is formally set to $\theta = +\infty$.

The cases when $\mathcal{I} \neq \emptyset$ and $\mathcal{I} = \emptyset$ lead to different outcomes of the attempt to change the value of the chosen nonbasic variable x_q .

- 1 If $\mathcal{I} \neq \emptyset$ then $\theta < +\infty$. By setting $x_q = \theta$ all basic variables remain feasible. Furthermore, as $\beta_p(\theta) = \beta_p - \theta \alpha_q^p = 0$, the p -th basic variable becomes zero, a property of nonbasic variables. Therefore, we can perform a basis change. Namely, the p -th basic vector can leave the basis and its place can be taken by \mathbf{a}_q . This change results in a neighboring basis as all other basic variables remain in the basis. Using θ as steplength in (2.35), the new solution is $\bar{\beta} = \beta(\theta) = \beta - \theta \alpha_q$, or componentwise

$$\bar{\beta}_i = \beta_i - \frac{\beta_p}{\alpha_q^p} \alpha_q^i, \quad \text{for } i = 1, \dots, m, \quad i \neq p. \quad (2.40)$$

The p -th basic variable becomes 0. It leaves the basis and is replaced by x_q as the new basic variable in position p with a value of

$$\bar{\beta}_p = \theta. \quad (2.41)$$

The new basis is

$$\bar{\mathcal{B}} = \mathcal{B} \setminus \{k_p\} \cup \{q\}, \quad (2.42)$$

where k_p is the original index of the p -th basic variable, see (2.6). We say that x_q is the *incoming variable* and x_{k_p} is the *outgoing variable* (or *entering* and *leaving variable*, respectively). As a result of the basis change, q becomes the new k_p . The replacement of the p -th basic variable of \mathbf{B} can be expressed by $\bar{\mathbf{B}} = \mathbf{B} + (\mathbf{a}_q - \mathbf{a}_{k_p})\mathbf{e}_p^T$. Similarly, $\bar{\mathbf{c}}_{\bar{\mathcal{B}}}^T = \mathbf{c}_{\mathcal{B}}^T + (c_q - c_{k_p})\mathbf{e}_p^T$.

Using the definition of $\mathbf{x}_{\mathcal{B}}(t)$ in (2.31) the value of the objective function can be expressed as a function of t ,

$$\begin{aligned} z(t) &= \mathbf{c}_{\mathcal{B}}^T \mathbf{x}_{\mathcal{B}}(t) + tc_q \\ &= \mathbf{c}_{\mathcal{B}}^T (\mathbf{B}^{-1}\mathbf{b} - t\mathbf{B}^{-1}\mathbf{a}_q) + tc_q \\ &= \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1}\mathbf{b} - t (\mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1}\mathbf{a}_q - c_q) \\ &= z(0) + td_q. \end{aligned} \quad (2.43)$$

Since $z = z(0)$, the new value of the objective function can be obtained by an update formula if we substitute θ for t in (2.43)

$$\bar{z} = z(\theta) = z + \theta d_q. \quad (2.44)$$

Verbally, the new value of the objective function is equal to the old one plus the steplength times the reduced cost of the selected variable.

The very simple formula in (2.44) deserves a little bit of studying as it highlights several things. First of all, it reaffirms that the nonnegative displacement of the incoming variable necessitates that $d_q < 0$. If $\theta > 0$ the objective improves, $\bar{z} < z$. On the other hand, $d_q < 0$ itself cannot guarantee a decrease in the objective function since the θ steplength can be zero if one or more of the β_i values are zero for $i \in \mathcal{I}$ (i.e., these basic variables are at the boundary of their feasibility range). In such a case the solution also remains unchanged by virtue of (2.40) and (2.41). This situation carries the danger of making a sequence of nonimproving basis changes and arriving back to an already encountered basis which can lead to an endless looping through this set of bases. This phenomenon is known as *cycling*. Obviously, cycling can, but does not necessarily, occur if the basis is degenerate. On the other hand, a degenerate basis can become nondegenerate during subsequent iterations. There are theoretical and practical safeguards to avoid cycling. More details are discussed later.

A large negative d_q may be thought of as a good choice for the entering variable. Practice, however, shows that if d_q is large then all the coefficients (2.32) in the transformed column are also large. Therefore, in the (2.39) ratio test the denominators tend to be large numbers, thus the ratios can be small. As a result, the progress can also be small. Ideally, q should be determined such that it gives the largest $|\theta d_q|$ product. This is called the principle of *largest progress*. However, the computational effort to find q in this way can be prohibitively large in case of sizeable problems.

To obtain the inverse of $\bar{\mathbf{B}}$ we have to update \mathbf{B}^{-1} . From (2.18), (2.20) and (2.21), the $\boldsymbol{\eta}$ vector of the elementary transformation matrix \mathbf{E} can be written as

$$\eta^p = \frac{1}{\alpha_q^p}, \quad \text{and } \eta^i = -\alpha_q^i \eta^p, \quad \text{for } i = 1, \dots, m, i \neq p. \quad (2.45)$$

With this $\boldsymbol{\eta}$, \mathbf{E} is defined as in (2.24). Finally, the update of the inverse is

$$\bar{\mathbf{B}}^{-1} = \mathbf{E} \mathbf{B}^{-1}. \quad (2.46)$$

- 2 If $\mathcal{I} = \emptyset$ then the (2.31) solution remains feasible for any large $t > 0$ as there is no blocking variable. From (2.43), the value of the objective function can be decreased infinitely. It means that in this case the *solution is unbounded*.

2.2.3 Algorithmic description of the simplex method

The developments in the preceding subsections make it possible to give a step-by-step algorithmic description of the simplex method for the LP problem of the form

$$\begin{aligned} & \text{minimize} && z = \mathbf{c}^T \mathbf{x} \\ & \text{subject to} && \mathbf{A} \mathbf{x} = \mathbf{b}, \\ & && \mathbf{x} \geq \mathbf{0}, \end{aligned} \tag{2.47}$$

if a feasible basis $\mathcal{B} = \{k_1, \dots, k_m\}$ is known. It is to be remembered that \mathbf{A} contains a unit matrix as we are dealing with CF-1 now. For reasons explained later in this chapter the algorithm to be presented is called *primal simplex method*.

Primal Simplex Method #1 (PSM-1)

Step 0. Initialization. Find an initial feasible basis \mathbf{B} .

Determine \mathbf{B}^{-1} , $\beta = \mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1} \mathbf{b}$ and $z = \mathbf{c}_{\mathcal{B}}^T \mathbf{x}_{\mathcal{B}}$.

Step 1. Compute simplex multiplier $\pi^T = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1}$.

Step 2. Check optimality (also known as *Choose Column* or *PRICE*): compute reduced costs $d_j = c_j - \pi^T \mathbf{a}_j$, for $j \in \mathcal{R}$. If all $d_j \geq 0$ (they satisfy the optimality condition), exit with “optimal solution found”. Otherwise, there is at least one variable with $d_j < 0$. Usually there are several. Choose one from among the candidates using some (simple or complex) criterion. The subscript of the selected variable is denoted by q .

Step 3. Transform the column vector \mathbf{a}_q of the improving candidate using (2.32): $\alpha_q = \mathbf{B}^{-1} \mathbf{a}_q$.

Step 4. Choose row (also known as *CHUZRO* or *pivot step* or *ratio test*): Determine set \mathcal{I} according to (2.38): $\mathcal{I} = \{i : \alpha_q^i > 0\}$. If $\mathcal{I} = \emptyset$ conclude that “solution is unbounded” and exit. Otherwise perform the (2.39) ratio test to obtain θ and determine basic position p of the outgoing variable x_{k_p} .

Step 5. Update

Solution, according to (2.40) and (2.41):

$$\begin{aligned}\bar{\beta}_i &= \beta_i - \theta \alpha_q^i, \quad \text{for } i = 1, \dots, m, \quad i \neq p, \\ \bar{\beta}_p &= \theta.\end{aligned}$$

Basis: $\bar{\mathcal{B}} = \mathcal{B} \setminus \{k_p\} \cup \{q\}$, as in (2.42).

Objective value, as given in (2.44): $\bar{z} = z(\theta) = z + \theta d_q$.

Basis inverse, by (2.45) and (2.46):

First, form $\boldsymbol{\eta}$ from α_q with components

$$\eta^p = \frac{1}{\alpha_q^p}, \quad \text{and } \eta^i = -\alpha_q^i \eta^p, \quad \text{for } i = 1, \dots, m, \quad i \neq p,$$

then \mathbf{E} , according to (2.22): $\mathbf{E} = [\mathbf{e}_1, \dots, \mathbf{e}_{p-1}, \boldsymbol{\eta}, \mathbf{e}_{p+1}, \dots, \mathbf{e}_m]$.

Finally apply (2.46) to determine the inverse of the new basis: $\bar{\mathbf{B}}^{-1} = \mathbf{E} \mathbf{B}^{-1}$.

Return to Step 1 with quantities with bar $\bar{\cdot}$ like $\bar{\mathcal{B}}$ replacing their respective originals, like \mathcal{B} .

PSM-1 is a logically correct theoretical algorithm. If the objective value strictly improves in every iteration no basis can be repeated. As the number of different bases is finite, PSM-1 will terminate in a finite number of steps with an answer to the problem (optimal or unbounded solution detected). Computational issues are not addressed in the above algorithmic description. They are discussed in Part II.

The usefulness of PSM-1 lies in the fact that it contains the most important ingredients of the more general versions of the simplex method in a clean form.

2.2.4 Finding a basic feasible solution

The most critical assumption of PSM-1 is the availability of a feasible basis. In some (rare) cases it is easy to find one. In general, however, it is an effort to obtain a feasible basis together with the corresponding BFS. In this subsection we present a method that, again, will be mostly of theoretical importance. However, it will serve as a starting point for the computationally more suitable procedures.

The problem is to find a BFS for problem (2.47) of section 2.2.3. It will be done in a ‘brute force’ way without any refinements. The computationally viable versions are presented in Part II of the book.

In the discussion it will be assumed that $\mathbf{b} \geq \mathbf{0}$ in (2.47). It can always be achieved by multiplying the offending rows by -1 . Now, a nonnegative *artificial variable* x_{n+i} is added to each left hand side of the constraints

$$\sum_{j=1}^n a_j^i x_j + x_{n+i} = b_i, \quad i = 1, \dots, m, \quad (2.48)$$

where $x_j \geq 0$ for $j = 1, \dots, n+m$. This is a different problem from the original one but it has the advantage that a basic feasible solution to it is directly available. Namely, the basis is the unit matrix of the artificial variables and the BFS is $x_{n+i} = b_i$ for $i = 1, \dots, m$.

(2.48) is equivalent to (2.47) if $x_{n+i} = 0$ for $i = 1, \dots, m$. Therefore, we can try to drive all artificial variables to zero through a series of basis changes performed within the framework of the simplex method. This can be achieved by solving the following LP problem:

$$\begin{aligned} \text{minimize } & z_a = \sum_{i=1}^m x_{n+i} \\ \text{subject to } & \sum_{j=1}^n a_j^i x_j + x_{n+i} = b_i, \quad i = 1, \dots, m, \\ & x_j \geq 0, \quad j = 1, \dots, n+m. \end{aligned} \quad (2.49)$$

This is an auxiliary problem to (2.47). Both have the same number (m) of constraints but in (2.49) the number of variables is $n+m$. An important difference is the objective function. Now it is the sum of the artificial variables. The original objective function is ignored at this stage.

We can use PSM-1 to solve the augmented problem in (2.49) by setting $\mathcal{B} = \{n+1, \dots, n+m\}$ and $\mathbf{B} = \mathbf{I}$. The initialization step of PSM-1 will ‘determine’ $\mathbf{B}^{-1} = \mathbf{I}$ and $\mathbf{x}_{\mathcal{B}} = \mathbf{b}$. The iterative steps of the algorithm can be carried out unchanged.

The theoretical minimum of (2.49) is zero, therefore, PSM-1 will always terminate with an optimal solution to the auxiliary problem. The optimal basis is denoted by \mathcal{B}^* and the optimal objective value by z_a^* .

If $z_a^* > 0$ then the constraints can only be satisfied if one or more artificial variables are positive. It entails that these constraints cannot be satisfied with the original variables at feasible level, in other words, the *problem is infeasible*.

If $z_a^* = 0$ then each artificial variable is equal to zero in one of the two possible ways: (i) it is nonbasic, or (ii) it is in the optimal basis at zero level. The index set of the basic positions of the latter variables is

denoted by \mathcal{D} . Our purpose is to identify a feasible basis to the original LP problem in (2.47). If case (i) holds for all artificial variables then all indices in \mathcal{B}^* are such that $1 \leq k_i^* \leq n$, therefore \mathcal{B}^* is a feasible basis to (2.47). PSM-1 can be applied to solve it starting with $\mathcal{B} = \mathcal{B}^*$ and using the original objective function.

If there are some variables in category (ii) the situation is a bit more complicated. Namely, if these variables are left in the problem (and in the basis) nothing prevents them from becoming positive during the subsequent iterations as the (true) objective function in (2.47) does not keep these variables under control. They can even be positive in an optimal solution found by PSM-1 which renders the result useless as the equivalence of the two problems does not hold.

A possible remedy is the elimination of the artificial variables from \mathcal{B}^* of (2.49). It can be achieved because the original matrix is of full row rank. As we are faced with case (ii) now, \mathcal{B}^* is a degenerate basis. The degree of degeneracy is at least $|\mathcal{D}|$. Pivoting on the positions of \mathcal{D} all artificials can be removed. During these steps the value of the objective function ($z_a = 0$) remains unchanged since the steplength will always be zero, see (2.39). For the same reason, the value of each basic variable also remains unchanged. Therefore, to remove a zero valued artificial variable from position p of the basis, any incoming variable x_q can be used provided it has $\alpha_q^p \neq 0$ (instead of the $\alpha_q^p > 0$ requirement of (2.39)).

As the α_q^p values are not available they have to be determined in order to be able to identify the incoming candidates. By definition,

$$\alpha_q^p = \mathbf{e}_p^T \boldsymbol{\alpha}_q = \mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{a}_q.$$

$\mathbf{e}_p^T \mathbf{B}^{-1}$ is nothing but the p -th row of \mathbf{B}^{-1} . Let us denote it by $\boldsymbol{\rho}^p$. Now α_q^p can be obtained by the dot product

$$\alpha_q^p = \boldsymbol{\rho}^p \mathbf{a}_q.$$

Based on these observations, the following procedure can be used to eliminate all zero valued artificial variables from \mathcal{B}^* .

Procedure Artificial Elimination

Perform the following three steps for all $p \in \mathcal{D}$.

Step 1. Compute $\boldsymbol{\rho}^p = \mathbf{e}_p^T \mathbf{B}^{-1}$.

Step 2. Take a nonbasic index q and compute $\alpha_q^p = \boldsymbol{\rho}^p \mathbf{a}_q$.

Step 3. If $\alpha_q^p \neq 0$ (or is of acceptable magnitude) perform a basis change by replacing the p -th basic variable by x_q as described in PSM-1. Otherwise, go back to Step 2 for a different nonbasic variable.

Because of the full row rank assumption of \mathbf{A} , at the end of this procedure we will have a feasible basis \mathcal{B} as in case (i) and PSM-1 can start in the same way as described there.

The main advantage of the above procedure of finding a BFS to the (2.47) LP problem is its simplicity. On the other hand, it is generally wasteful in terms of the introduced new variables. Namely, the logical variables (already present in (2.1) – (2.3)) of rows with $b_i \geq 0$ can be used instead of the artificials. They have the additional advantage that they do not have to be removed if they remain in the optimal basis of the (2.49) auxiliary problem. The efficiency of this method can substantially be improved by more sophisticated procedures. They will be discussed in Part II.

2.2.5 The two-phase simplex method

The (2.1) – (2.3) LP problem was solved in two phases. This approach (not the only imaginable one) is called the *two-phase simplex method*.

In phase-1 it looks for a feasible solution to the problem. It does it by creating and solving an auxiliary problem, the so called *phase-1 problem*. If the search is unsuccessful we can conclude that the problem is infeasible. If this is the case, the optimal solution of the phase-1 problem points out which constraints could not be satisfied. It does not necessarily mean that these constraints are the real cause of infeasibility. With other phase-1 methods different constraints may become unsatisfied. However, by any method, the fact of infeasibility remains true. Analyzing the real cause of infeasibility is an important modeling issue. Interested readers are referred to relevant results of Chinneck [Chinneck, 1997] and Greenberg [Greenberg, 1983].

Phase-2 starts when phase-1 successfully terminated with $z_a^* = 0$ and all artificial variables have been removed from the basis. The transition is seamless. Only the objective coefficients in the definition of d_j will be different from phase-1. No ‘preprocessing’ of the true objective function is needed. The phase-2 algorithm deals with alternative optima of phase-1 (they all have $z_a^* = 0$) and looks for an optimal solution to the original problem in (2.1) – (2.3). The algorithm may terminate with an optimal solution or with the indication that the solution is unbounded.

In case of the two-phase method we solve two, closely related, LP problems in succession by the same simplex algorithm.

2.3. Duality

Duality is a very general principle that has enormous importance in linear programming. Its simplest form can be stated in the following way. To any LP problem,

$$(P1) \quad \begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A} \mathbf{x} \geq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

which we call *primal LP*, there is an associated LP problem, called its *dual*, in the form of

$$(D1) \quad \begin{aligned} \max \quad & \mathbf{b}^T \mathbf{y} \\ \text{s.t.} \quad & \mathbf{A}^T \mathbf{y} \leq \mathbf{c}, \\ & \mathbf{y} \geq \mathbf{0} \end{aligned}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and all vectors are of compatible dimensions. Here, the dual objective function is $\mathbf{b}^T \mathbf{y}$. We also say that (P1) and (D1) constitute a primal-dual pair. There is a symmetry between the primal and the dual. The primal variables are associated with the columns of \mathbf{A} while the dual variables are associated with its rows. The objective and the right-hand side vectors also swap their roles in the primal and dual. From the above formulation it is easy to see that the dual of the dual is the primal. Namely, (D1) can be written as

$$\begin{aligned} -\min \quad & (-\mathbf{b}^T \mathbf{y}) \\ -\mathbf{A}^T \mathbf{y} \quad & \geq -\mathbf{c} \\ \mathbf{y} \quad & \geq \mathbf{0}. \end{aligned}$$

This is the form of (P1) with \mathbf{y} being the vector of unknowns. Therefore, we can apply the method of writing its dual (swapping the role of \mathbf{x} and \mathbf{y}) and obtain

$$\begin{aligned} -\max \quad & (-\mathbf{c}^T \mathbf{x}) \\ -\mathbf{A} \mathbf{x} \quad & \leq -\mathbf{b} \\ \mathbf{x} \quad & \geq \mathbf{0}, \end{aligned}$$

which is equivalent to

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \mathbf{A} \mathbf{x} \quad & \geq \mathbf{b} \\ \mathbf{x} \quad & \geq \mathbf{0}, \end{aligned}$$

that is the original (P1).

If the primal is given as

$$(P2) \quad \begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A} \mathbf{x} = \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

then the dual is

$$(D2) \quad \begin{aligned} \max \quad & \mathbf{b}^T \mathbf{y} \\ \text{s.t.} \quad & \mathbf{A}^T \mathbf{y} \leq \mathbf{c}. \end{aligned}$$

Here, the dual variables are unrestricted in sign (free variables). This form can be derived from the (P1)–(D1) relationship noting that an equality constraint $\mathbf{a}^i \mathbf{x} = b_i$ can be replaced by two inequalities, like $\mathbf{a}^i \mathbf{x} \geq b_i$ and $-\mathbf{a}^i \mathbf{x} \geq -b_i$, and remembering that a free variable can be written as a difference of two nonnegative variables as shown in subsection 1.2.1.1.

In addition to the formal correspondence, there exist some fundamental mathematical relationships between the primal and dual. Some of them are briefly discussed below.

THEOREM 2.4 (WEAK DUALITY) *Given an arbitrary primal feasible solution \mathbf{x} to (P2) and any dual feasible solution to (D2), then for the corresponding objective values relation $\mathbf{y}^T \mathbf{b} \leq \mathbf{c}^T \mathbf{x}$ holds.*

PROOF. As \mathbf{x} and \mathbf{y} are solutions of their respective problems and $\mathbf{x} \geq \mathbf{0}$, we have

$$\mathbf{b} = \mathbf{A} \mathbf{x} \quad \text{and} \quad \mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T.$$

Multiplying the left hand side equation by \mathbf{y}^T , the right-hand side inequality by \mathbf{x} and comparing the two, we obtain

$$\mathbf{y}^T \mathbf{b} = \mathbf{y}^T \mathbf{A} \mathbf{x} \quad \text{and} \quad \mathbf{y}^T \mathbf{A} \mathbf{x} \leq \mathbf{c}^T \mathbf{x},$$

which proves the theorem. □

In verbal terms, any feasible solution to the dual defines a lower bound on the primal objective. Similarly, any feasible solution to the primal represents an upper bound on the dual objective. The question is whether the two bounds can be equal? The answer is formulated in the following theorem.

THEOREM 2.5 (STRONG DUALITY) *If any problem of the primal-dual pair (P2) and (D2) has a feasible solution and a finite optimum then the same holds for the other problem and the two optimum values are equal.*

PROOF. Because of the symmetry of the primal-dual pair it is sufficient to prove just one direction of the statement.

Let \mathbf{B} denote an optimal basis to the primal problem and $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b} \geq \mathbf{0}$. As the optimality conditions are satisfied, from proposition 2.2.1 and (2.30) we have

$$\mathbf{c}^T \mathbf{B}^{-1} \mathbf{a}_j = c_j, \quad j \in \mathcal{B}, \quad (2.50)$$

$$\mathbf{c}^T \mathbf{B}^{-1} \mathbf{a}_j \leq c_j, \quad j \in \mathcal{R}. \quad (2.51)$$

\mathbf{A} can be assumed to be partitioned as $\mathbf{A} = [\mathbf{B} \mid \mathbf{R}]$. Therefore, if we define

$$\mathbf{y} = \mathbf{c}_B^T \mathbf{B}^{-1} \quad (2.52)$$

then relations (2.50) and (2.51) mean that \mathbf{y} satisfies the dual constraints, thus it is a feasible solution to (D2) (no sign restriction on \mathbf{y}).

The objective value of the primal is $\mathbf{c}_B^T \mathbf{x}_B = \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{b} = \mathbf{y}^T \mathbf{b}$, i.e., the two objective values are equal. By the weak duality theorem the dual objective can not be better than this common value, therefore it is optimal for the dual. \square

2.4. The dual simplex method

2.4.1 Dual feasibility

Let us consider the (P2)–(D2) pair. As (P2) is a special case of computational form #1, \mathbf{A} contains a unit matrix and $m < n$. With the introduction of vector $\mathbf{w} = [w_1, \dots, w_n]^T$ of dual logical variables (D2) can be rewritten as

$$\max \quad \mathbf{b}^T \mathbf{y} \quad (2.53)$$

$$\text{s.t.} \quad \mathbf{A}^T \mathbf{y} + \mathbf{w} = \mathbf{c}, \quad (2.54)$$

$$\mathbf{w} \geq \mathbf{0}. \quad (2.55)$$

Let \mathbf{B} be a basis to \mathbf{A} . It need not be primal feasible. Rearranging (2.54), we get $\mathbf{w}^T = \mathbf{c}^T - \mathbf{y}^T \mathbf{A}$, or in partitioned form

$$\mathbf{w}_B^T = \mathbf{c}_B^T - \mathbf{y}^T \mathbf{B}, \quad (2.56)$$

$$\mathbf{w}_{\mathcal{R}}^T = \mathbf{c}_{\mathcal{R}}^T - \mathbf{y}^T \mathbf{R}. \quad (2.57)$$

The nonnegativity (and, in this case, the feasibility) requirement (2.55) of \mathbf{w} in partitioned form is $[\mathbf{w}_B^T, \mathbf{w}_{\mathcal{R}}^T]^T \geq \mathbf{0}$. Choosing $\mathbf{y}^T = \mathbf{c}_B^T \mathbf{B}^{-1}$ we obtain

$$\mathbf{w}_B^T = \mathbf{c}_B^T - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{B} = \mathbf{0}, \quad (2.58)$$

$$\mathbf{w}_{\mathcal{R}}^T = \mathbf{c}_{\mathcal{R}}^T - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{R} = \mathbf{d}_{\mathcal{R}}^T \geq \mathbf{0}, \quad (2.59)$$

where \mathbf{d}_R denotes the vector formed by the primal reduced costs of the nonbasic variables, see (2.27). Since (2.58) is satisfied with any basis and \mathbf{y} is unrestricted in sign, a basis \mathbf{B} is dual feasible if it satisfies (2.59). This is, however, nothing but the primal optimality condition, see (2.30). Therefore, we can conclude that the dual feasibility condition is equivalent to the primal optimality condition. Additionally, the dual logicals are equal to the primal reduced costs. Therefore, w_j and d_j can be used interchangeably.

This observation suggests the following idea. Let us assume we have a dual feasible basis to (2.53) – (2.55). If it is also primal feasible then it is optimal for both. If not then make basis changes that lead to primal feasibility while maintaining dual feasibility. The next section investigates how this idea can be turned into an algorithm.

2.4.2 Improving a dual feasible solution

Let us consider the primal-dual pair in the following form. The primal

$$(P3) \quad \min \quad z = \mathbf{c}^T \mathbf{x} \quad (2.60)$$

$$\text{s.t.} \quad \mathbf{A} \mathbf{x} = \mathbf{b}, \quad (2.61)$$

$$\mathbf{x} \geq \mathbf{0}, \quad (2.62)$$

and its dual, using \mathbf{d} instead of \mathbf{w} ,

$$(D3) \quad \max \quad Z = \mathbf{b}^T \mathbf{y} \quad (2.63)$$

$$\text{s.t.} \quad \mathbf{A}^T \mathbf{y} + \mathbf{d} = \mathbf{c}, \quad (2.64)$$

$$\mathbf{d} \geq \mathbf{0}. \quad (2.65)$$

\mathbf{A} is assumed to contain a unit matrix in accordance with CF-1.

Let \mathbf{B} be a feasible basis to the dual problem. If it is not primal feasible then it is not optimal. Therefore, there exists a neighboring dual feasible basis with a better (or not worse) dual objective value unless the dual problem is unbounded. There are several possibilities to derive the rules of such a basis change. We present one that is suitable for a straightforward generalization.

Note that in (2.56) the basic equations are satisfied as equalities so that the corresponding dual logicals are equal to zero, see (2.58). We can try to relax one of the equations, say the p -th, and compensate it by changing the dual variables so that, in the meantime, the dual objective improves.

The p -th dual basic equation corresponds to variable x_{k_p} . For convenience, we introduce notations $\mu = k_p$ and $\beta_p = x_{k_p}$. Formally, the p -th dual basic equation changes from $\mathbf{a}_\mu^T \mathbf{y} = c_\mu$ to $\mathbf{a}_\mu^T \mathbf{y} \leq c_\mu$. We say this

constraint is relaxed negatively because $\mathbf{a}_\mu^T \mathbf{y}$ is decreased. If the change in the p -th dual equation is parameterized by $t \leq 0$ then the basic dual equations are maintained if the dual variables are adjusted as

$$\mathbf{B}^T \mathbf{y}(t) - t \mathbf{e}_p = \mathbf{B}^T \mathbf{y}, \quad (2.66)$$

where $\mathbf{y}(t)$ denotes the dual solution as a function of t . Multiplying both sides of this equation by \mathbf{B}^{-T} (the inverse of \mathbf{B}^T) and denoting the p -th column of \mathbf{B}^{-T} by $\boldsymbol{\rho}_p$ the dual solution as a function of t can be expressed as

$$\mathbf{y}(t) = \mathbf{y} + t \boldsymbol{\rho}_p. \quad (2.67)$$

The corresponding dual objective value is

$$Z(t) = \mathbf{b}^T \mathbf{y}(t) = \mathbf{b}^T \mathbf{y} + t \mathbf{b}^T \boldsymbol{\rho}_p = \mathbf{b}^T \mathbf{y} + t \beta_p = Z(0) + t \beta_p. \quad (2.68)$$

As t moves away from 0 negatively the rate of change in the dual objective is β_p , which leads to an improvement over $\mathbf{b}^T \mathbf{y}$ if $\beta_p < 0$. This suggests that a basic equation with

$$\beta_p < 0 \quad (2.69)$$

(an infeasible primal basic variable) has to be selected for relaxation.

The dual logicals can also be expressed as a function of t . Namely, from (2.58) and (2.59), using d_j instead of w_j , as already suggested,

$$d_j(t) = c_j - \mathbf{a}_j^T \mathbf{y}(t), \quad j \in \mathcal{R}. \quad (2.70)$$

Substituting $\mathbf{y}(t)$ from (2.67) and taking into account that $\mathbf{y}^T = \mathbf{c}_B^T \mathbf{B}^{-1}$ and $\boldsymbol{\rho}_p^T = \mathbf{e}_p^T \mathbf{B}^{-1}$, we obtain

$$\begin{aligned} d_j(t) &= c_j - \mathbf{a}_j^T (\mathbf{y} + t \boldsymbol{\rho}_p) \\ &= c_j - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{a}_j - t \mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{a}_j \\ &= d_j - t \alpha_j^p, \end{aligned} \quad (2.71)$$

because $d_j = c_j - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{a}_j$ and $\alpha_j = \mathbf{B}^{-1} \mathbf{a}_j$. Obviously, $d_j(0) = d_j$. The nonnegativity requirement is

$$d_j(t) = d_j - t \alpha_j^p \geq 0. \quad (2.72)$$

From this formula it is clear that if $\alpha_j^p \geq 0$ then $w_j(t)$ remains feasible for any $t \leq 0$. If $\alpha_j^p < 0$ then (2.72) gives $t \geq d_j / \alpha_j^p$. Therefore, the largest displacement of t for which (2.72) holds for every nonbasic j , and

which we denote by θ_D , is determined by positions with $\alpha_j^p < 0$. If we define

$$\mathcal{J} = \{j : \alpha_j^p < 0, j \in \mathcal{R}\} \quad (2.73)$$

then it is easy to see that the *dual steplength* is

$$\theta_D = \frac{d_q}{\alpha_q^p} = \max_{j \in \mathcal{J}} \left\{ \frac{d_j}{\alpha_j^p} \right\}. \quad (2.74)$$

Here, q denotes a nonbasic position for which the maximum is attained and α_q^p is called the pivot element, just as in the primal simplex. θ_D may not be uniquely defined. Expression in (2.74) is also referred to as *dual ratio test*. To be able to perform it the primal reduced costs (d_j -s) and the α_j^p coefficients of the transformed pivot row are needed.

If $t = \theta_D$ then the reduced cost of x_q becomes zero, a feature of the basic variables. Therefore, it can replace the basic variable (x_{k_p}) of the relaxed constraint in the basis. If this replacement is performed solutions also change. The new dual solution can be obtained from the update formulas in (2.67) and (2.71) by substituting $t = \theta_D$. The value of the logical variable of the relaxed dual equation is $\bar{d}_q = -\theta_D$, see (2.66). Obviously, $\bar{d}_q \geq 0$.

The new value of the dual objective can be determined from (2.68) by substituting $t = \theta_D$ into $Z(t)$

$$\bar{Z}_D = Z(\theta_D) = Z(0) + \theta_D \beta_p. \quad (2.75)$$

Updating the primal solution can be done exactly the same way as in the case of the primal simplex, see (2.40) and (2.41). While the necessary α_q^i components are available for the primal (because of the ratio test) they have to be computed from their definition of $\alpha_q = \mathbf{B}^{-1} \mathbf{a}_q$ in the dual.

The new basis is $\bar{\mathcal{B}} = \mathcal{B} \setminus \{k_p\} \cup \{q\}$ with q becoming the new k_p .

If $\alpha_j^p = 0$ for some $j \in \mathcal{R}$ then we have *dual degeneracy*. It makes $\theta_D = 0$ if such a j occurs in \mathcal{J} and, as a result, the objective and the solution remain unchanged. Only the basis changes. Just as in primal, dual degeneracy may lead to cycling. However, there are theoretical (and also practical) remedies to avoid it, to be discussed in Part II.

2.4.3 Algorithmic description of the dual simplex method

The step-by-step algorithmic description of the dual simplex method is based on the developments of the previous section. It can be used to

solve an LP problem of the form

$$\begin{aligned} & \text{minimize} && \mathbf{c}^T \mathbf{x} \\ & \text{subject to} && \mathbf{Ax} = \mathbf{b}, \\ & && \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

if a dual feasible basis $\mathcal{B} = \{k_1, \dots, k_m\}$ is known. \mathbf{A} contains a unit matrix as we are dealing with CF-1.

Algorithm: Dual Simplex Method #1 (DSM-1)

Step 0. Initialization.

Determine \mathbf{B}^{-1} , $\boldsymbol{\beta} = \mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}\mathbf{b}$, $\mathbf{y}^T = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1}$, $\mathbf{d}_{\mathcal{R}}^T = \mathbf{c}_{\mathcal{R}}^T - \mathbf{y}^T \mathbf{R}$ and $Z = \mathbf{y}^T \mathbf{b}$.

Step 1. Check optimality (also known as *dual PRICE*): If $\boldsymbol{\beta} \geq \mathbf{0}$ (current solution is primal feasible), exit with “optimal solution found” Otherwise, there are one or more basic variables with $x_{k_i} < 0$. Choose one from among the candidates using some (simple or complex) criterion. Its basic position is denoted by p , the outgoing candidate is x_{k_p} .

Step 2. Compute $\boldsymbol{\alpha}_{\mathcal{R}}^p = \mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{R}$ which is a vector containing the non-basic components of the *updated pivot row*. It is done in two stages. First, extract the p -th row of \mathbf{B}^{-1} (the p -th column of \mathbf{B}^{-T}) by $\boldsymbol{\rho}^p = \mathbf{e}_p^T \mathbf{B}^{-1}$ then compute $\alpha_j^p = \boldsymbol{\rho}^p \mathbf{a}_j$ for $j \in \mathcal{R}$.

Step 3. Dual ratio test. Determine set \mathcal{J} as in (2.73): $\mathcal{J} = \{j : \alpha_j^p < 0, j \in \mathcal{R}\}$. If $\mathcal{J} = \emptyset$ conclude that “dual solution is unbounded” and exit. Otherwise perform the (2.74) dual ratio test to determine the maximum ratio θ_D and the incoming variable x_q .

Step 4. Update

Dual solution:

$$\begin{aligned} \bar{\mathbf{y}} &= \mathbf{y}(\theta_D) = \mathbf{y} + \theta_D (\boldsymbol{\rho}^p)^T, \\ \bar{d}_j &= d_j - \theta_D \alpha_j^p, \quad j \in \mathcal{R}, \text{ and } \bar{d}_{k_p} = -\theta_D. \end{aligned} \quad (2.76)$$

Primal solution:

Compute $\boldsymbol{\alpha}_q = \mathbf{B}^{-1} \mathbf{a}_q$ and $\theta = \beta_p / \alpha_q^p$.

Perform update according to (2.40) and (2.41):

$$\begin{aligned} \bar{\beta}_i &= \beta_i - \theta \alpha_q^i, \quad \text{for } i = 1, \dots, m, \quad i \neq p, \\ \bar{\beta}_p &= \theta. \end{aligned}$$

Objective value, as given in (2.75): $\bar{Z} = Z(\theta_D) = Z + \theta_D \beta_p$.

Basis: $\bar{\mathcal{B}} = \mathcal{B} \setminus \{k_p\} \cup \{q\}$ with q becoming the new k_p .

Basis inverse, the same way as in primal, by (2.45) and (2.46):

Form $\boldsymbol{\eta}$ from $\boldsymbol{\alpha}_q$ with components

$$\eta^p = \frac{1}{\alpha_q^p}, \quad \text{and } \eta^i = -\alpha_q^i \eta^p, \quad \text{for } i = 1, \dots, m, \quad i \neq p.$$

Determine \mathbf{E} , according to (2.22):

$$\mathbf{E} = [\mathbf{e}_1, \dots, \mathbf{e}_{p-1}, \boldsymbol{\eta}, \mathbf{e}_{p+1}, \dots, \mathbf{e}_m].$$

Apply (2.46) to determine the inverse of the new basis: $\bar{\mathbf{B}}^{-1} = \mathbf{E}\mathbf{B}^{-1}$.

Return to Step 1 with quantities with bar $\bar{\cdot}$ like $\bar{\mathbf{B}}^{-1}$ replacing their respective originals, like \mathbf{B}^{-1} .

If dual degeneracy is treated properly no basis is repeated. Therefore, DSM-1 will terminate in a finite number of iterations giving an answer to the problem (optimal solution or dual solution unbounded).

Looking at DSM-1 it can be concluded that the dual simplex method actually works on the primal problem but performs the basis change using different rules. It first determines the outgoing and then the incoming variable. While the primal ratio test ensures that the solution remains primal feasible the dual ratio test does the same for the dual.

An interesting feature of DSM-1 is that the primal and dual objective values are equal in every iteration. It comes from $Z = \mathbf{y}^T \mathbf{b} = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{b} = \mathbf{c}_{\mathcal{B}}^T \mathbf{x}_{\mathcal{B}} = z$. Of course, while \mathbf{y} is always dual feasible, $\mathbf{x}_{\mathcal{B}}$ is feasible only when an optimal solution is reached.

The same question arises here as in the primal. How do we get a dual feasible basis? In some cases it is very simple. For example, if the objective coefficients of the structural variables in (2.60) are all nonnegative then the unit matrix of logical variables is a dual feasible basis. Some methods have been worked out for the case when such an obvious choice is not available. The most typical of them is the dual big-M method. In this case an extra ' \leq ' type constraint $\sum_{j=1}^n x_j + x_{n+1} = M$, with $x_{n+1} \geq 0$, is added to the original constraints. M can be chosen large enough so that this constraint is practically non-binding. It is possible to develop an algorithm that gradually builds up a dual feasible basis if one exists. Details of this method can be found, e.g., in [Padberg, 1995]. It is not reproduced here as a more general method will be discussed in Chapter 10.

The dual simplex algorithm was introduced by Lemke [Lemke, 1954] in 1954. In the early years of its discovery it was not considered a serious alternative to the primal simplex. This was mostly due to its ‘under-developed’ status compared to the primal method with respect to advanced algorithmic and computational techniques. Nowadays the dual simplex method is algorithmically well developed and is a full-blown counterpart of the primal. Details of the enhancements are discussed also in Chapter 10 of this book.

2.4.4 Further properties of the primal-dual relationship

The first part of the strong duality theorem says that if any problem of the primal-dual pair has a feasible solution and a finite optimum so does the other. The next question is what can be said about the dual if the primal is (i) feasible but the objective is unbounded, or (ii) infeasible? The following theorem establishes the answer to these questions.

THEOREM 2.6 *If the primal (P2) has a feasible solution but the objective value is not bounded from below then its dual pair, (D2), has no feasible solution.*

PROOF. Let us assume the contrary, i.e., the dual has a feasible solution \hat{y} . Because of the weak duality theorem, $c^T x \geq b^T \hat{y}$ holds for all primal solutions with this \hat{y} which is impossible since $c^T x$ can be made arbitrarily small due to its unboundedness. \square

Because of the symmetry between the primal and its dual pair, a similar statement for the dual is also true. Namely, if the dual has a feasible solution but its objective value is not bounded from above then its primal pair has no feasible solution.

It is important to remember the direction of implication of the theorem: if a problem is unbounded then its counterpart has no feasible solution. The converse is not necessarily true, i.e., from the infeasibility of one of the problems it does not follow that the other is unbounded. Below we give an example where both primal and its dual are infeasible.

EXAMPLE 2.1 *Both primal and dual are infeasible.*

Primal:

$$\begin{aligned} \min \quad & 3x_1 + 5x_2 \\ \text{s.t.} \quad & x_1 + 2x_2 = 3 \\ & 2x_1 + 4x_2 = 1 \\ & x_1, x_2 \text{ free.} \end{aligned}$$

Dual:

$$\begin{aligned} \max \quad & 3y_1 + y_2 \\ \text{s.t.} \quad & y_1 + 2y_2 = 3 \\ & 2y_1 + 4y_2 = 5 \\ & y_1, y_2 \text{ free.} \end{aligned}$$

Both constraint sets are obviously contradictory.

If one of the problems is infeasible then the only thing we can say with certainty is that the other has no optimal solution.

There is an interesting theorem by Clark which says that unless both primal and dual are infeasible at least one of them has an unbounded feasibility set.

We can summarize the conclusions of this section in the following tabular form:

Primal unbounded	\implies	Dual infeasible.
Dual unbounded	\implies	Primal infeasible.
Primal infeasible	\implies	Dual $\begin{cases} \text{infeasible or} \\ \text{unbounded.} \end{cases}$
Dual infeasible	\implies	Primal $\begin{cases} \text{infeasible or} \\ \text{unbounded.} \end{cases}$

2.5. Concluding remarks

There are several possibilities to present the primal and dual simplex algorithms. The way we followed in this chapter has lead us to a version called the *Revised Simplex Method (RSM)*, originally developed by Dantzig and Orchard-Hays [Dantzig and Orchard-Hays, 1953]. It will be pointed out in later chapters that this is the only version that is a candidate for solving large scale problems. RSM is in some contrast to Dantzig's original version which is often referred to as *tableau version* of the simplex. Nowadays, the latter has mostly educational importance. First courses in mathematical programming use it because it is very suitable for the introduction of the main concepts of linear programming and the simplex method, c.f., [Hillier and Lieberman, 2000].

The tableau version maintains a fully transformed \mathbf{A} matrix which can otherwise be obtained by $\mathbf{B}^{-1}\mathbf{A}$ if we know \mathbf{B}^{-1} . After every basis change it updates this matrix by \mathbf{E} . Since in computational forms #1 and #2 there is a unit matrix present in \mathbf{A} the transformed \mathbf{A} contains \mathbf{B}^{-1} itself, therefore, \mathbf{B}^{-1} gets automatically updated. It follows that updating the full tableau is computationally more expensive than updating \mathbf{B}^{-1} in RSM. On the other hand, $\mathbf{B}^{-1}\mathbf{A}$ contains all updated

rows and columns including those (which are unknown at the beginning of an iteration) that turn out to be needed for the primal or dual steps.

There are, however, other differences between the RSM and the version using the tableau form. As any form of the simplex method requires the use of floating point arithmetic, computational errors may accumulate that can lead to a wrong answer to the LP problem. RSM has a natural error control capability. Namely, if \mathbf{B}^{-1} is updated too many times it may lose accuracy. In this case it is possible to recompute it from the defining columns. This procedure eliminates the contribution of columns to the updated form of \mathbf{B}^{-1} that have left the basis. Additionally, the basis inverse can be determined as a product of elementary transformation matrices. This *product form of the inverse (PFI)* is not unique and the inherent flexibility can be utilized to generate good quality inverses. Criteria of the quality and details of PFI are discussed in Chapter 8. The tableau version of the simplex method does not have this error control mechanism.

COMPUTATIONAL TECHNIQUES

Chapter 3

LARGE SCALE LP PROBLEMS

LP models of real-life problems tend to be large in size. It is a consequence of the effort to create more realistic models. This activity is supported by increasingly capable modeling systems. They are mainly commercial products and their discussion falls outside the scope of this book.

The big challenge with the simplex method is to design and implement computational versions of it that can solve large scale and complex LP problems reliably and efficiently. Before analyzing the requirements of large scale optimization we have to analyze the characteristics of large scale problems. In brief, they are: sparsity, structure, fill-in and susceptibility to numerical troubles. This chapter gives a short overview of these features.

First of all, it is to be noted that the actual meaning of ‘large scale’ is constantly changing. What was a large problem in the 1960s is considered tiny at the turn of the century. While in the former case the solution of an LP problem with several hundreds of rows was an achievement, forty years later problems with tens of thousands of rows are solved routinely.

The purposes of this chapter are well served if the linear programming problem is considered as being defined by the tuple $(\mathbf{A}, \mathbf{b}, \mathbf{c}, \boldsymbol{\ell}, \mathbf{u})$, where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{c}, \boldsymbol{\ell}, \mathbf{u} \in \mathbb{R}^n$, and $\boldsymbol{\ell}$ and \mathbf{u} contain individual bounds on variables.

3.1. Sparsity

The size of an LP problem is naturally determined by the size of \mathbf{A} in terms of the number of its elements (determined by the number of rows and columns). With the introduced notation, it is mn . Matrix

elements are usually stored as 8-byte double precision numbers in a computer. Therefore, the explicit storage of \mathbf{A} requires $8mn$ bytes. For an \mathbf{A} with 10,000 rows and 20,000 columns (not a particularly large problem) it amounts to 200 million elements that require 1.6 Gigabytes of storage. This is an exceedingly large number. Additionally, it is not only the storage but also the number of operations per iteration in this magnitude that is prohibitive.

Large scale LP problems, however, have an important and typical feature, namely, they contain relatively few nonzero entries. This feature is called *sparsity*, sometimes also referred to as *low density*. Let ν denote the number of nonzeros in an entity (matrix or vector). The density of matrix \mathbf{A} is defined as the ratio of the nonzeros to the total number of positions

$$\varrho(\mathbf{A}) = \frac{\nu_A}{mn}.$$

Here, the \mathbf{A} in the argument indicates that the density and the number of nonzeros are those of \mathbf{A} . The density of a vector is defined in a similar sense. For example,

$$\varrho(\mathbf{b}) = \frac{\nu_b}{m}$$

denotes the density of the RHS vector \mathbf{b} . When it is obvious from the context the entity to which the density is referred to can be omitted. ϱ is often given as percentage rather than a fraction.

An interesting observation shows that, on the average, there are 5–10 nonzeros per column independent of the size of the problem. It implies that larger problems are increasingly more sparse. According to this ‘rule’ the average density of problems with around one thousand rows is not more than 1% and density drops below 0.1% for problems with ten thousand rows.

In any case, to characterize the size of large scale problems, it is necessary to include the number of nonzeros of \mathbf{A} . In year 2000 a problem with $m = 2,000$, $n = 10,000$ and $\nu_A = 40,000$ is considered small, while a problem with the same m and n but with $\nu_A = 200,000$ is rather of medium size since, very likely, it will require substantially more computational effort to solve it. N.B., the density of the first problem is 0.2% while that of the second is 1.0%.

Kalan [Kalan, 1971] noticed that in many cases even among the nonzero entries there are only few distinct values. This phenomenon is called *super sparsity*. For instance, it is typical that there are many ± 1 entries in a matrix. Obviously, super sparsity is not the superlative of sparsity. It is simply an additional characterization of the nature of the nonzero elements. In some cases it is advantageous to utilize super sparsity to reduce memory requirements by storing each distinct value

only once and using appropriate pointers to access them. The importance of this idea was very high in the 1960s and 1970s when the size of computer memory was typically not more than 1MB.

As an example consider the following vector and matrix:

$$\mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ -3 \\ 1 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} -1 & 0 & 0 & 2 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 4 & -1 \\ 0 & 0 & 0 & -3 & 1 \end{bmatrix} \quad (3.1)$$

In this case, $\varrho(\mathbf{v}) = 3/5 = 0.6$ (or 60%) and $\varrho(\mathbf{A}) = 10/25 = 0.4$ (or 40%). Among the ten nonzeros of \mathbf{A} there are five distinct values $\{-1, 1, 2, -3, 4\}$.

3.2. Instances of large problems

There are several publicly available collections of large scale real-life LP problems. The most widely known is in NETLIB [Gay, 1985]. These problems are used by researchers to test the results of algorithmic developments and by users who want to compare the performance of different solvers. The information file of NETLIB provides problem statistics for each problem that include m , n and ν_A , in the first place.

For an illustration, we show an excerpt from the `readme` file of NETLIB problems in `netlib/lp/data`. The heading of the first four columns are self-explanatory while the meaning of the remaining columns is as follows. `Bytes` is the size of the compressed file containing the problem in MPS format (discussed in Chapter 6), `BR` indicates whether there are individual bounds (`B`) or range (`R`) constraints in the problem, `Optimal Value` is the verified optimal value.

PROBLEM SUMMARY TABLE

Name	Rows	Cols	Nonzeros	Bytes	BR	Optimal Value
25FV47	822	1571	11127	70477		5.5018458883E+03
80BAU3B	2263	9799	29063	298952	B	9.8723216072E+05
BOEING1	351	384	3865	25315	BR	-3.3521356751E+02
CZPROB	930	3523	14173	92202	B	2.1851966989E+06
D2Q06C	2172	5167	35674	258038		1.2278423615E+05
DEGEN3	1504	1818	26230	130252		-9.8729400000E+02
GREENBEA	2393	5405	31499	235711	B	-7.2462405908E+07
FIT2D	26	10500	138018	482330	B	-6.8464293294E+04
FIT2P	3001	13525	60784	439794	B	6.8464293232E+04
MAROS	847	1443	10006	65906	B	-5.8063743701E+04

MAROS-R7	3137	9408	151120	4812587		1.4971851665E+06
PILOT87	2031	4883	73804	514192	B	3.0171072827E+02
STOCFOR3	16676	15695	74004	(see NOTES)		-3.9976661576E+04

There is an additional set of larger LP problems in the `kennington` subdirectory of `netlib/lp/data`. To obtain a feel about the sizes that can appear in LP, we show the statistics of some of the problems in this set. Excerpts are taken from the relevant `readme` file which has a slightly different column structure than `lp/data`.

Name	rows	columns	nonzeros	bounds	MPS	optimal value
CRE-B	9649	72447	328542	0	10478735	2.3129640e+07
CRE-D	8927	69980	312626	0	9964196	2.4454970e+07
KEN-13	28633	42659	139834	85318	8254122	-1.0257395e+10
KEN-18	105128	154699	512719	309398	29855000	-5.2217025e+10
OSA-60	10281	232966	1630758	0	52402461	4.0440725e+06
PDS-10	16559	48763	140063	16148	5331274	2.6727095e+10
PDS-20	33875	105728	304153	34888	11550890	2.3821659e+10

In column `bounds` the total number of lower and upper bound coefficients is given, `MPS` indicates the uncompressed size of the MPS file of the problem in bytes.

3.3. Structure

Large scale LP problems often carry some structure, like the ones that represent dynamic models, spatial distribution or uncertainty. Such problems contain several, often identical, blocks that are just loosely connected by some constraints and/or variables having nonzero coefficients in the rows/columns of several blocks. Structure can be utilized in at least two different ways in the interest of obtaining more capable solution techniques. One of them results in the *decomposition algorithms* while the other one leads to the substantial enhancement of several steps of the simplex method. In Part II we will concentrate on the latter.

Structure can be visualized by displaying the nonzero pattern of the `A` matrix. There exist several software tools that enable such visualization. Depending on the level of sophistication of the tool, the magnitudes of the elements in the nonempty cells can be seen represented by color coded dots, user can zoom in on a particular area of the matrix up to the level when the actual numerical values of the matrix are displayed, etc. These viewers are very useful tools that can reveal unknown properties of LP models and also can suggest good solution strategies. Figures 3.1 and 3.2 show the nonzero structure of problems PILOT87 and GREENBEA, respectively.

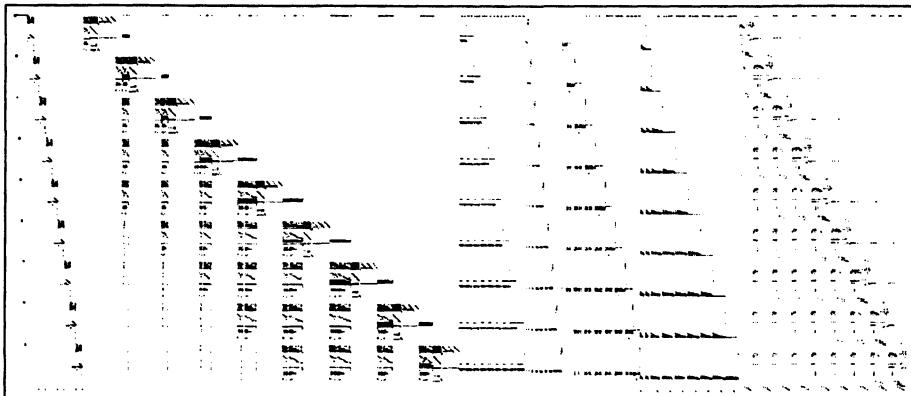


Figure 3.1. Nonzero pattern of PILOT87.MPS. Problem size: 2031 rows (including the objective function), 4883 columns, 73804 nonzeros.

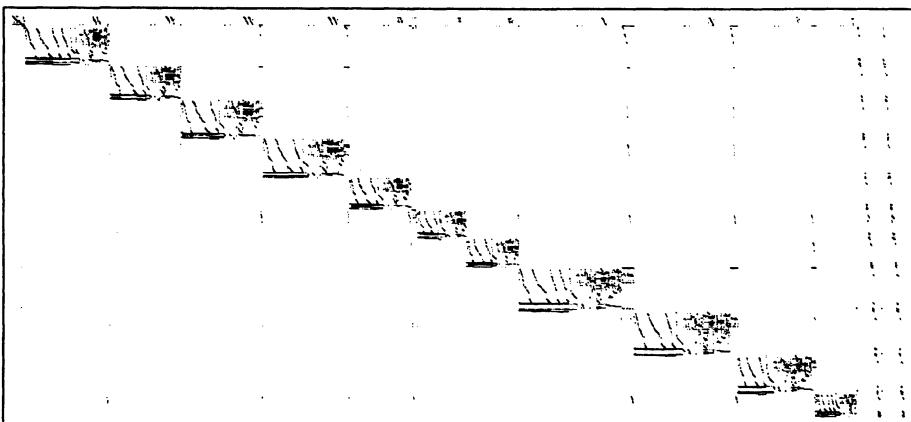


Figure 3.2. Nonzero pattern of GREENBEE.MPS. Problem size: 2393 rows (including the objective function), 5405 columns, 31499 nonzeros.

3.4. Fill-in

If the tableau form of the simplex method is used it is updated by an elementary transformation matrix \mathbf{E} after every basis change. Though the matrices of large scale problems are sparse, they tend to fill in. It means that, as a result of premultiplying (the already transformed) \mathbf{A} by \mathbf{E} , nonzeros can appear in places of zeros.

As \mathbf{E} is the most important updating factor it is worth looking at how it transforms a column vector which, for simplicity, is denoted now by \mathbf{a} .

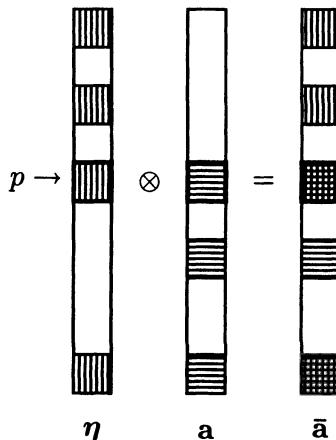


Figure 3.3. Creation of nonzeros during update. $\eta \otimes a$ stands for the **Ea** operation such that η is the p -th column of **E**, as in (3.2). In this example two new nonzeros have been created (the vertically hatched boxes in the upper part of \bar{a}).

If the η vector is in position p in **E** then $\mathbf{Ea} = \bar{\mathbf{a}}$ expands to

$$\begin{bmatrix} 1 & \eta^1 \\ \ddots & \vdots \\ \eta^p & \\ \vdots & \ddots \\ \eta^m & \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_p \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} a_1 + a_p\eta^1 \\ \vdots \\ a_p\eta^p \\ \vdots \\ a_m + a_p\eta^p \end{bmatrix}. \quad (3.2)$$

This equation has the message that if $a_p = 0$ then $\bar{\mathbf{a}} = \mathbf{a}$. In other words, no operation is needed to obtain the result. In this way, a simple logical test can save many memory accesses and (otherwise trivial but still time consuming) arithmetic operations. The practical importance of this simple fact is enormous. If \mathbf{a} is a large and sparse vector there is a very good chance that a_p is zero, therefore, it need not be updated with this (and probably many other) elementary transformation matrix.

On the other hand, if $a_p \neq 0$ then the nonzero pattern of $\bar{\mathbf{a}}$ is the union of the nonzero patterns of η and \mathbf{a} (except in the very rare case of cancellation). More formally, if the index set of nonzero positions of η and \mathbf{a} are denoted by \mathcal{E} and \mathcal{A} , respectively, then the nonzero pattern of $\bar{\mathbf{a}}$ is $\mathcal{E} \cup \mathcal{A}$. The number of newly created nonzeros is thus $|\mathcal{E} \setminus \mathcal{A}|$. Therefore, if η is quite dense and $a_p \neq 0$ then many new nonzeros can be created in one step. Figure 3.3 demonstrates how fill-ins are generated.

Fill-in can be so heavy that storing, and performing operations on, the transformed **A** becomes prohibitive.

One of the advantages of the revised simplex method is that it does not require the maintenance of the transformed tableau. Any updated row or column can be reproduced by \mathbf{B}^{-1} when needed. However, \mathbf{B}^{-1} is updated by the same \mathbf{E} which causes the fill-in. As a result, \mathbf{B}^{-1} also tends to become denser after every update. The real challenge is to find an equivalent form of \mathbf{B}^{-1} which uses much fewer nonzeros to represent the inverse. This form, as already hinted, is the product form of the inverse and its variants, most of all the *elimination form of the inverse (EFI)*. With EFI we can observe, for instance, that the inverse of a $10,000 \times 10,000$ basis with 50,000 nonzeros can be represented by around 150,000 nonzeros instead of the potentially maximum 100 million nonzero entries of \mathbf{B}^{-1} .

3.5. Numerical difficulties

The solution of large scale linear programs requires very substantial computational effort. Since floating point arithmetic has to be used (with finite precision, of course) all sorts of numerical troubles can occur. While it is not true that all large problems are numerically difficult, the tendency is that they are more prone to running into troubles than smaller problem instances.

The sources of inaccuracies are the operations with the basis inverse. During the $\mathbf{c}^T \mathbf{B}^{-1}$ and $\mathbf{B}^{-1} \mathbf{a}$ type operations both rounding and cancellation errors can occur and accumulate. Errors propagate from one iteration to the next one by using elementary transformation matrices of type \mathbf{E} for updating. The updating cycle is roughly the following. The column of the incoming variable is updated by \mathbf{B}^{-1} (omitting subscript q) as $\boldsymbol{\alpha} = \mathbf{B}^{-1} \mathbf{a}$. This $\boldsymbol{\alpha}$ is used to create the nontrivial column $\boldsymbol{\eta}$ of \mathbf{E} . Finally, \mathbf{E} is used to update \mathbf{B}^{-1} as $\bar{\mathbf{B}}^{-1} = \mathbf{E} \mathbf{B}^{-1}$ and also the solution $\bar{\boldsymbol{\beta}} = \mathbf{E} \boldsymbol{\beta}$. Additionally, other quantities are also computed or updated using \mathbf{B}^{-1} and important decisions are made depending on the sign or magnitude of computed quantities (can the solution be improved, is the pivot element large enough, etc.).

In case of the revised simplex method there is a natural way to reinstate accuracy by periodically recomputing the inverse of the basis from its defining columns. This is an important mechanism and the main contributing algorithmic technique that makes it possible to solve large and difficult LP problems.

Numerical troubles manifest themselves in several different ways, like a numerically inaccurate solution, a wrong optimal basis, or even a wrong answer to the linear program (e.g., feasible problem declared infeasible, or optimal problem found unbounded). It is not only the final outcome that can be affected by numerical inaccuracies. Computations can ‘blow

up' at any time if a small element was wrongly chosen as a pivot when it was actually a numerical garbage instead of the algebraically correct zero. It usually leads to a singular basis (which is actually not a basis), can cause a jump back to infeasibility after a reinversion, just to mention a few of the possible adverse outcomes.

The susceptibility of the simplex method to numerical troubles has motivated considerable research in algorithmic techniques that are numerically more stable and also efficient. In this respect the most remarkable progress has been achieved in handling the basis inverse or other equivalent forms that can be used to provide the necessary information needed for the simplex method. As a result, the techniques used for inverting or factorizing the basis are able to produce numerically stable and, at the same time, very sparse representations (inverse in product form or LU factors). Details are discussed in Chapter 8.

Chapter 4

DESIGN PRINCIPLES OF LP SYSTEMS

Solving any reasonably sized LP problem cannot be done without the use of a computer. Therefore, theoretical algorithms, in our case versions of the simplex method, have to be translated into executable computer programs. This activity is called the *implementation* of an algorithm. Over the years techniques and methods have been developed in support of this activity. This new knowledge is often referred to as *implementation technology* of optimization software.

It is not uncommon that unsophisticated implementation of a theoretically correct algorithm causes surprises when used for solving problems. It can happen that the program

- works at very low speed,
- has a huge memory requirement,
- does not converge,
- gives wrong answer (e.g., declares a problem unbounded or infeasible though it has an optimal solution),
- works only on small test problems (worse: just on a specific test problem),
- sometimes does not work at all (e.g., breaks down due to numerical difficulties),
- and many more.

Implementing an optimization algorithm is a complex issue. It requires deep knowledge of the (i) theory of optimization algorithms,

(ii) computational linear algebra, (iii) relevant results of computer science, (iv) advanced techniques and methods of software engineering and (v) clear understanding of some important details of the ever evolving computer hardware technology. These are the five main ingredients of the successful development of optimization software.

This chapter first overviews the requirements of LP software, then briefly discusses the most important features of hardware and software environments that have relevance to LP systems. The purpose of the chapter is to lay down the foundations and highlight the motivation of the forthcoming algorithmic techniques. The discussion in this chapter relies on the paper of Maros and Khaliq [Maros and Khaliq, 2002]. It is to be noted that the pioneering work in the area of implementation technology of optimization software was done by Orchard-Hays [Orchard-Hays, 1968, Orchard-Hays, 1978].

4.1. Requirements of LP software

As the development of LP software is a major effort it is reasonable to expect that the codes possess some desirable properties. The most important ones can briefly be summarized as follows.

Robustness: A wide range of problems can be solved successfully. If the attempt is unsuccessful a meaningful answer is returned (e.g., unrecoverable numerical troubles). Robustness implies reliability and accuracy.

Efficiency: In simple terms, efficiency means execution speed. A fast algorithm implemented with code optimized by good design and profiling permits a solution to be obtained quickly.

Capacity: Large and difficult (e.g., mixed integer) problems can be solved on a given computer configuration.

Portability: The system can be used on different hardware and software platforms. *Binary portability* can be achieved if the same or a similar hardware/software combination is used. *Source level portability* is easier to achieve and can be done by using some of the standard high level languages that are available on nearly all types of computers (independent of hardware and operating system).

Memory scalability: The system can adjust to the size of the main memory and operate reasonably efficiently if at least a minimum amount of memory is available. (The system is not critically memory-hungry.) At the same time, it can adapt well if more memory is allocated.

Modularity: The implementation is divided into data structures, functions, and translation units according to a design that is both intuitive and adheres to good software engineering practice. Units perform specific tasks with low coupling to other units.

Extensibility: The code permits both the simple addition of new features internally to the product and the straightforward integration of that product as a building block for other code.

Code stability: The behavior of the optimization system code is well-defined even in an error situation. At the least, the system will not leak resources under an exception condition, and should attempt to continue running safely and meaningfully.

Maintainability: The impact of changes or additions to the code is minimal. Source code is well annotated.

User friendliness: Sometimes also referred to as *ease of use*. This soft notion has several interpretations and has a different meaning for a beginner and an advanced user. Under this heading the availability of a stand-alone and callable subroutine version is also understood.

These properties can be ensured partly during algorithm design and partly during the implementation phase as shown in the following table.

	Algorithm design	Implementation
Reliability	x	x
Accuracy	x	x
Efficiency	x	x
Capacity	x	x
Portability		x
Modularity		x
Code stability		x
Maintainability		x
Scalability	x	x
User friendliness	x	x

It is clear from this table that the role of implementation in creating a general purpose solver is absolutely crucial. The orders of magnitude in differences of a good and an average implementation simply cannot be expressed because it is not only the speed that counts. It is really robustness that end-users expect from a system so that they can use it

in a ‘black box’ fashion. And in this sense the differences between implementations can be enormous depending on how much of the advanced computational techniques of the simplex method have been included.

4.2. Computer hardware

While the hardware component of the computing environment changes continually there are some stable elements. Having said that, it is still true that this is the most volatile section of the whole book. What is covered in this section reflects the status quo at the time of writing (August 2001). Though readers a decade later may find the current state quite outdated the knowledge of some techniques and ideas will still be helpful in understanding the computational aspects of optimization.

For LP software the hardware features of the memory hierarchy and the central processing unit (CPU, or processor) of a computer are the most important. Knowledge of how to exploit these technologies now, and directions they will take in the future, is crucial in the production of a portable code that not only executes quickly today, but also scales well onto future machines.

Over the years, processors have become so fast that now memory access is the bottleneck in the speed of numerical computing. To overcome this problem a hierarchy of memory components has been developed. Knowing this technicality can help design better algorithmic elements for the simplex method.

Memory hierarchy, in decreasing order of speed, decreasing order of cost per memory unit, and increasing capacity is organized as follows: registers, L1 cache, L2 cache, main memory (RAM), and the hard disk.

Each CPU has a small set of registers which are best used for rapidly changing and/or heavily accessed variables. Unfortunately, registers are not addressable by languages other than machine-specific assembler, but efficient code coupled with an optimizing compiler can give good register usage (e.g., allocation of loop variables).

The L1 cache resides on a CPU, is marginally slower than a register, and can be either a unified instruction and data, or a separate instruction block and a data block to hold segments of code and data, respectively. The slower L2 cache has recently come to exist in the CPU too, with a size of 256KB or more compared with 32–128KB for the L1 cache. The caches keep data close to the CPU for rapid access, otherwise that data would be fetched on demand from slower main memory. A frequently accessed variable held in a cache exhibits *temporal locality*. Each time the CPU wants a variable, the cache is queried and if the data is there we have a *cache hit*, with the item supplied to the CPU from the cache location. Since contiguous data tends to be accessed in programs, a

cache will insert not only data currently needed, but also some items around that too to get more cache hits by guessing it will need what is nearby—this is *spatial locality*. If data is not consistently accessed, is widely separated from other data, or is requested for the first time, we have the risk of a *cache miss*. For a direct-mapped cache, data is retrieved from memory, a location is isolated in the cache and before it is placed there any existing item is ejected from the cache. Fully associative and set-associative caches have the capacity to allow several items to be held at a cache location, but they are slower than the direct-mapped cache due to scanning operations that must be performed to try to get a cached item or signal a cache miss. Cache misses can delay a CPU for hundreds of cycles, but they are easily caused, such as through not properly organized sparse matrix computations.

RAM and the hard disk drive have the slowest access times, but sizes typically in hundreds of megabytes, and many tens of gigabytes, respectively. RAM works at the nanosecond level, yet currently CPUs are so fast that the RAM slows execution! Hard disk drives give persistent storage, but work at the millisecond level. Code that leads to thrashing in virtual memory mechanisms or needs considerable disk access clearly will have poor execution times.

The features of the CPU that most affect optimization software are: the type of the processor, the number of *floating point units* (FPUs) and *arithmetic logic units* (ALUs), and the number and depth of pipelines.

A processor is typically described as a *Complex Instruction Set Computer* (CISC) or a *Reduced Instruction Set Computer* (RISC), with a given clock frequency and instruction bit width. CISC CPUs have the advantage of low memory requirements for code, but although RISC machines have higher memory demands, they are faster due to hardware optimization of key operations. CISC-RISC hybrid processors are available that wrap a CISC shell over a RISC core. Recently, reconfigurable processors are in production, the Crusoe chip being an example, that have the property of being able to work as CISC or RISC depending on the code they are given at the expense of some CPU speed. Many factors, such as the system bus speed, affect the speed of a machine. However, in general a higher clock frequency means a faster processor, as does a higher bit width of instructions.

Floating point (FP) calculations on a processor occur in the FPUs, built to implement the IEEE 754 standard usually. Double precision (DP) numbers, typically 8 bytes in length (around 16 decimal digits accuracy), are processed here, but it should always be remembered that calculations are subject to errors that can accumulate. Complementing FPUs are ALUs which undertake exact arithmetic on integers only, and

sometimes they are employed for address calculations. The more FPUs and ALUs a chip has, the greater the throughput of number processing that can be attained.

Most CPUs now have at least one pipeline made up of about 5–10 structural units, designed to perform an operation and pass the result to the next unit. Superscalar architectures have multiple pipelines. A pipeline can exist in many parts of a CPU, such as the ALU or the FPU. The depth, however, is fixed by factors such as the format of instructions and the amount of instruction unrolling to be done if the pipeline has to be stopped. A pipeline increases CPU performance by allowing a sequence of instructions to be in the CPU per clock cycle, thereby boosting instruction throughput. A pipeline entity, however, is efficient only when the results of one operation are not pending on the results of another—this would cause a *stall*. *Branching*—jumps from boolean evaluation, function calls, and function returns—in a program is a major cause of these halts in the running of pipelines.

A CPU with a pipeline can stall as one option in a problem situation, but other options are *branch prediction* or *dynamic branching*. Branch prediction makes an assumption, e.g., a boolean test is always false, and bypasses a stall. If the guess was wrong, then operations have to be ‘rewound’ and the correct instructions used. In dynamic branching, a small hardware lookup table is used holding the depth of branching and the number of times a branch held true or false. By using the lookup table a pipeline can be adjusted to follow the branch that occurs most often—this is useful in looping constructs since the wrong decision occurs only at the final iteration. The trouble with this method is that too many branches in code will overfill the table. The branching problem is one of several issues significant enough for the number of pipeline units to be kept low.

In software, the best method to utilize pipelines is to avoid branching, especially nested, as much as possible. Loop unrolling is a well-known technique to speed code by reducing boolean tests. This technique is implemented by optimizing compilers when the number of iterations is known at compile time. If the number of iterations is known only at run time, then code safety issues arise that necessitate employing loop unrolling with great caution. If simple branching can be avoided by a single operation in code, then it is preferable to replace the branch.

Since branching can be caused by function call and return, reducing the number of functions in code is useful. As an example, in C++, the keyword `inline` allows compilers to check the syntax of a short function and then embed its code in place of all function calls. This is one way to approach a trade-off in dealing with branching in code:

a program should be modular and maintainable, but also must exhibit minimal branching. Sometimes, it is better to leave branching in place, such as when branches call different functions or alter different data types, since program transformations can leave the code unreadable. If possible, placing all of the disjoint code of a function that has had simple branching removed into one location can be of use, as streamlined execution occurs once the branch direction has been found by the CPU.

There is an interesting development in hardware technology under the heading of *reconfigurable computing*. This notion refers to hardware reconfiguration as opposed to a software one (see Crusoe processor above). Boolean satisfiability problems can directly be represented by reconfigurable computing units so that the evaluation of a Boolean expression is carried out in hardware at very high speed. For each problem instance the hardware is reconfigured (the equivalent of traditional ‘problem input’). Early indications show that a speedup of up to 100 times can easily be achieved by this method. Investigations are underway on how to exploit this powerful new tool for better computational algorithms in integer programming.

4.3. The software side of implementing LP solvers

The LP algorithms are converted into computer programs by using certain software tools. Optimization programs are written in some high level language, mostly FORTRAN or C/C++ (in year 2001). These languages are highly standardized so their code can easily be compiled on different hardware/software platforms resulting in identically working programs. This is the basis of source level portability.

The efficiency of optimization code is affected by the following software factors: the operating system, the choice of programming language, and the compiler employed.

Operating systems act as the interface between user software and hardware, typically allowing multiple users access to resources in a way that gives the impression that each user has sole access. This latter feature—known as *multitasking*—is achieved today through intelligent scheduling of processes or threads to a processor. Operating systems differ markedly from each other and are very likely the most changing elements of the software environment of optimization software.

The operating system best suited for an optimizer is difficult to isolate early. It is of great benefit therefore that coding be undertaken in a standardized language, so that through portability the best operating system will be found by compiling and testing on different platforms.

The choice of programming language has an enormous impact on optimization software. The main contenders in this field are versions of

FORTRAN(77, 90, 95), C, and C++. A language decision must consider at least the following: standardization so that portability is allowed (the aforementioned have ANSI standards), whether the language is interpreted (like APL and MatLab) or compiled (like the main contenders above) since the former is excellent for prototyping ideas but the latter are the means for fast code. Additional important issues are whether there is control of static and dynamic memory (FORTRAN77 is capable only of the former; FORTRAN90 and later versions have dynamic memory support just as C/C++), whether code can be made modular for extensibility (e.g., the ability to create not only a stand-alone optimizer, but the same optimizer as a callable routine that can be integrated into a library), and the amount of safety the language provides against erroneous execution (e.g., the exception safety mechanism of C++).

To illustrate a decision for a programming language for an optimization system, we take the example of C++. The language is standardized to ANSI/ISO level since 1997 with good compiler support on practically all platforms. It is a compiled language so we can expect well-written code to have good execution speed on different systems. Memory management is under the control of the programmer, so we can allocate and deallocate resources at run time, e.g., using functions `new` and `delete` or standard components such as vectors or queues. Object-oriented programming is allowed using classes so that modular code can be made that is intuitive to expand and maintain. Last of all, there is a well-defined exception mechanism in the language that allows software to recover from errors without program termination which could leak resources.

The quality of a compiler influences the speed of well-written code on a given platform. A code may use generic speed optimizations—such as low function depth to reduce excessive stack usage, use pointers instead of arrays or vice versa, and exhibit minimal branching—but because the optimizing ability of each compiler is different, the run time performance is also different. What must be noted, however, is that the quality of compiler-optimized code is competitive with or exceeds hand-optimized code, and the former is improving. For example, C code optimized by the GNU C compiler `gcc` is very difficult to better with optimization by hand. Languages like C++ allow the inclusion of assembler language code into high level code which the compiler then integrates. In this respect hand-optimized code may better compiler-optimized code. However, impressive gains in using assembler will need to be balanced against the loss of portability of a system. Lastly, compilers differ in the quality and breadth of the libraries they are packaged with. This can lead to speed differences in the same function that is part of a language standard, e.g., the `malloc` function for allocating memory in C.

In the light of the above it is not surprising that an ‘environment aware’ code of an optimization algorithm can be, without loss of portability up to three times faster than an unsophisticated but otherwise carefully coded version of exactly the same algorithm.

Chapter 5

DATA STRUCTURES AND BASIC OPERATIONS

In Chapter 3 it was pointed out that matrix \mathbf{A} of the constraints of large scale LP problems is generally very sparse. The columns contain, on the average, 5 – 10 nonzeros independent of the row dimension of the matrix. As a basis \mathbf{B} is part of \mathbf{A} it has a similar sparsity. Probably the only difference between the sparsity of the two is that \mathbf{A} usually has a structure of the nonzeros while in \mathbf{B} the columns appear intermixed without any particular structure.

Sparsity is the single most important feature of large scale LP problems that make it possible to process them in computers. A fundamental issue to the success of implementing the simplex method is how the sparse entities (matrices, vectors) are stored and used in the solution algorithm. This raises the need for carefully designed data structures and operations.

There is no one data structure that is the best for all purposes. In fact, data structures must be designed with a consideration of the operations to be performed on the entities. The general framework of designing proper data structures for sparse vectors and matrices and procedures of using them is often referred to as *sparse computing*. In this chapter we present the most important techniques of sparse computing that bear relevance to the implementation of the simplex method. It is common to call the sparsity exploiting version of the SM *sparse simplex method* which is sometimes referred to by the acronym SSX.

Two main types of data structures are distinguished. *Static* data structures remain unchanged during the computations. A typical example is matrix \mathbf{A} of the LP constraints. The storage requirement of static structures can be determined in advance and they remain fixed. *Dynamic* data structures keep changing in size and their storage require-

ments cannot be anticipated. For example, this type of data structures are needed during inversion/factorization of the basis where the fill-in is not known in advance. Sometimes the boundary between these two types is blurred. For instance, if the LP matrix undergoes a preprocessing (resulting in temporary deletion of rows and columns) the originally static structure is subject to a one-off change. Therefore, it must be designed to enable efficient preprocessing so that the static nature of the structure is preserved.

It is not a surprise that dynamic data structures are more complicated and accessing individual data items in such a structure is more time consuming than that of with static data structures. However, in SSX both types have their well established role. Therefore, in this chapter both of them will be discussed and demonstrated.

In addition to the obvious need for storing the problem data it is equally important to set up proper structures for storing and accessing extra information, like type and status of variables (basic, feasible, etc.), individual lower and upper bounds, the basic vectors, basic solution, and much more. Their storage requirement is not negligible. The extra information has to be ‘close’ to the entity where it belongs to allow for better utilization of the cache memory. Interestingly, this issue has a noticeable impact on the speed of the solver. Therefore, it must be given a serious consideration.

5.1. Storing sparse vectors

Sparse vectors can be stored in *full-length*, also called *explicit form*, as a dense vector where all zeros are explicitly represented. While it is a rather wasteful method it has several advantages. It is simple, access to the elements is straightforward (through an index of an array) and the storage requirements are known in advance. Furthermore, operations on dense vectors are easy to design.

As a better way, only the nonzeros of a sparse vector are stored as (*integer*, *double*) pairs (i, v_i) , for $i \in \mathcal{Z}$, where \mathcal{Z} is the set of indices of nonzeros in vector \mathbf{v} . In a computer program we usually keep a separate integer and a double array each of length of the number of entries (at least). This is the *compressed* or *packed* storage of a sparse vector.

Vector \mathbf{v} of the example in (3.1), $\mathbf{v} = [1, 0, 0, -3, 1]^T$, can be stored with $\mathcal{Z} = \{1, 4, 5\}$ and $|\mathcal{Z}| = 3$ as

len	3
ind	1 4 5
val	1.0 -3.0 1.0

where `len` is the length of the vector in terms of nonzeros, `ind` is the array of indices and `val` is the array of values of the nonzero entries of the stored vector.

The storage is said to be *ordered* if the subscripts are in a monotone (usually ascending) order, like in this example, otherwise *unordered*. Usually, there is no need to maintain ordered form of sparse vectors (see later).

Sometimes an explicit vector has to be converted into packed form. This operation is called *gather*. The opposite operation is *scatter* when a packed vector is expanded into full-length representation.

5.2. Operations involving sparse vectors

Most operations involving sparse vectors are best performed if one of the participating vectors is scattered into a work array w . All components of this array need to be initialized to zero. If the size of w is large the frequent initialization can consume considerable processor time. Fortunately, as computations are performed we always can keep track of positions involved in the operations. Therefore, those (generally few) positions that become nonzero can easily be restored to zero at the completion of the operation. So the message is clear, avoid initializing large work arrays in every iteration as much as possible because much initialization can be very expensive.

After this simple but very important preliminary remark, we discuss the two most important operations occurring in SSX.

5.2.1 Addition and accumulation of sparse vectors

A typical operations in the simplex method is the updating of a sparse vector by adding/subtracting to/from it a multiple of another sparse vector. In some cases a series of sparse vectors are involved in updating.

Assume the two vectors are $\mathbf{u}, \mathbf{v} \in \mathbb{R}^m$ and the operation in question is

$$\mathbf{u} := \mathbf{u} + \lambda \mathbf{v} \quad (5.1)$$

with some scalar λ . If both vectors are in unordered packed form then the direct performance of this operation (without unpacking) would require a search of the vectors to identify the matching pairs and the newly created nonzeros (fill-ins). A better way is the following. Assume there is a full-length work vector (array) $\mathbf{w} \in \mathbb{R}^m$ available with every position initialized to zero. We work with this array, modify its elements, but at the end restore it to zero. We show two alternatives of how the operation can be organized.

One possibility is Method-1 that can be described in the following way.

Step 1. Scatter \mathbf{v} into \mathbf{w} .

Step 2. Scan the indices of the nonzeros of \mathbf{u} . For each entry u_i , check w_i . If it is nonzero, update u_i ($u_i := u_i + \lambda w_i$) and reset w_i to zero.

Step 3. Scan the indices of the nonzero entries of \mathbf{v} . For each entry v_i check w_i .

If w_i is nonzero, we have a fill-in: $u_i := \lambda v_i$. Add the new component of \mathbf{u} to the data structure of \mathbf{u} and reset w_i to zero,

otherwise, do nothing because $w_i = 0$ is a result of a fill-in and restoration to zero in Step 2 .

If the vectors are in ordered form scan can be made without expanding the vector but any fill-in (which is usually put to the end of the data structure) can destroy ordering in one step. This is one of the reasons why we do not assume ordering in packed form. The other is that nearly all operations can be performed efficiently without this assumption.

Below we present a pseudo-code for Method-1. It is assumed that the nonzeros of \mathbf{u} are represented in the two arrays indu and valu and the similar holds for \mathbf{v} in indv and valv . The number of nonzeros in the vectors (their lengths in terms of nonzeros) is denoted by nzu and nzv , respectively. The mathematical \neq symbol is represented by \neq .

```

do j = 1,nzv
  w(indv(j)) = valv(j)           /* scatter into w */
enddo
do j = 1,nzu
  i = indu(j)
  if (w(i) != 0.0) then
    valu(j) = valu(j) + lambda * w(i)      /* update */
    w(i) = 0.0
  endif
enddo
do j = 1,nzv
  i = indv(j)
  if (w(i) != 0.0) then
    nzu = nzu + 1
    valu(nzu) = lambda * w(i)           /* fill-in */
    indu(nzu) = i
    w(i) = 0.0
  endif
enddo

```

```

    endif
enddo

```

There is another way of performing (5.1). Here, again, we assume the availability of an explicit working vector $\mathbf{w} = \mathbf{0}$. Method-2 can be described as follows.

Step 1. Scatter \mathbf{u} into \mathbf{w} .

Step 2. Scan \mathbf{v} . For each entry v_i check w_i . If it is nonzero, update w_i ($w_i := w_i + \lambda v_i$), otherwise set $w_i := \lambda v_i$ and add i to the data structure of \mathbf{u} (fill-in).

Step 3. Scan the modified data structure of \mathbf{u} . For each i , set $u_i := w_i$, $w_i := 0$.

As with the previous case, at the end of this algorithm \mathbf{w} is restored to zero and can be used for similar purposes again.

In the pseudo-code of Method-2 we use the same assumptions as in Method-1.

```

do j = 1,nzu
    w(indu(j)) = valu(j)                      /* scatter into w */
enddo
do j = 1,nzv
    i = indv(j)
    if (w(i) != 0.0) then
        w(i) = w(i) + lambda * valv(j)          /* update */
    else
        w(i) = lambda * valv(j)                  /* fill-in */
        nzu = nzu + 1
        indu(nzu) = i
    endif
enddo
do j = 1,nzu
    i = indu(j)
    valu(j) = w(i)
    w(i) = 0.0
enddo

```

Method-2 is slightly more expensive than the first one because the revised \mathbf{u} is stored first in \mathbf{w} and then placed in \mathbf{u} . However, if the following sequence of operations is to be performed

$$\mathbf{u} := \mathbf{u} + \sum_j \lambda_j \mathbf{v}_j,$$

then Method-2 is more efficient. The reason is that Steps 1 and 3 are performed only once and Step 2 is repeated for $\mathbf{v}_1, \mathbf{v}_2, \dots$, with intermediate results for \mathbf{u} remaining in \mathbf{w} .

Both algorithm share the following very important common feature: *the number of operations to be performed is a function of the number of nonzeros and is completely independent of the explicit size (m) of the participating vectors.*

5.2.2 Dot product of sparse vectors

The dot product of two m dimensional vectors \mathbf{u} and \mathbf{v} is defined as $\mathbf{u}^T\mathbf{v} = \sum_{i=1}^m u_i v_i$. With full-length vectors this operation is logically very simple. However, if the vectors are sparse, a large number of multiplications and additions may be performed with zero valued operands which is wasteful for two reasons. First, the operands need to be fetched (memory access through cache hits and misses) and second, performing double precision arithmetic operations with result known to be zero.

If \mathbf{u} and \mathbf{v} are stored as sparse vectors, the best to do is to scatter one of them and perform the dot product between a sparse and a full-length vector. Assuming that work array \mathbf{w} has been initialized to zero, the following pseudo-code will compute the dot product if the nonzeros of \mathbf{u} and \mathbf{v} are represented in the same way as before.

```

do j = 1, nzu
    w(indu(j)) = valu(j)           /* scatter into w */
enddo
dotprod = 0.0
do i = 1, nzv
    dotprod = dotprod + valv(i) * w(indv(i))
enddo
do j = 1, nzu
    w(indu(j)) = 0.0               /* restore zeros */
enddo

```

As nzu is usually much smaller than m (the full size of \mathbf{u}) the ‘extra’ loop at the end which restores the zeros in \mathbf{w} is a great saving over a the reinitialization of the entire array.

5.3. Storing sparse matrices

A straightforward way to store a sparse matrix is to store it as a collection of sparse column (row) vectors. The nonzeros of the columns (rows) can be stored consecutively in the same array, indices (row indices of entries in $rind$) and elements (in val) separately, of course. The starting position (beginning) of each column (say, $cbeg$) must be recorded. If we

have n columns, we need $n + 1$ such positions, the last one pointing to the first position after the last element of the last column. In this way the length of column j (in terms of nonzeros) is $\text{clen}(j) = \text{cbeg}(j+1) - \text{cbeg}(j)$. In general, clen is not needed explicitly. However, if the columns are not stored consecutively, it is necessary to maintain clen . In this case a column is accessed by cbeg and clen . Keeping clen makes the structure more flexible. Alternatively, the end of columns $\text{cend}(j)$ can be used which gives a similar flexibility.

For convenience, we repeat matrix **A** of example (3.1) and use it for demonstrating the ideas of storing sparse matrices.

$$\mathbf{A} = \begin{bmatrix} -1 & 0 & 0 & 2 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 4 & -1 \\ 0 & 0 & 0 & -3 & 1 \end{bmatrix} \quad (5.2)$$

Table 5.1 shows how matrix **A** can be represented as a set of sparse column vectors. It is to be noted that some of the column vectors are unordered, that is, the indices of the elements are not in an ascending order. Under subscript 6, we enclosed 11 in parentheses to indicate that this entry of cbeg is needed only if clen is not used.

Table 5.1. Matrix **A** stored as a collection of sparse column vectors.

Subscripts	1	2	3	4	5	6	7	8	9	10
cbeg	1	3	4	5	8	(11)				
clen	2	1	1	3	3					
rind	3	1	3	2	4	1	5	3	4	5
val	1.0	-1.0	-1.0	-1.0	4.0	2.0	-3.0	2.0	-1.0	1.0

The data structure of this representation is static. Its main problem becomes evident when a new matrix element needs to be inserted or an existing element deleted.

Deletion of a column entry can actually be achieved in a simple but a bit wasteful way. If the element is spotted it can be swapped with the last element in the column (order of elements is unimportant) and the length of the column reduced by one. It is possible only if the more flexible storage scheme with the length of columns is used. If, at a later stage, the column is enlarged by the insertion of a new element the space of the deleted element can be reused.

In case of insertion the data structure needs to be used dynamically. If there is some free space left after each column the new elements can be placed there, if not, a fresh copy of the column has to be created and placed after the last up-to-date column leaving its out-of-date version unaltered. This action is possible only if the array has some ‘elbow room’ at the end. (Since columns may not be consecutive anymore, both `cbeg` and `clen` are needed to access them.) Obviously, there is a danger of running out of the allocated array. Therefore, from time to time the vectors may have to be shifted up to form a consecutive sequence from the beginning again to reuse the space of out-of-date copies of columns. This operation is called *compression*, more specifically, this is column compression.

The main use of this data structure is to store **A**, in particular the columns of the structural variables. Once the data structure of the matrix has been set up it remains unchanged during the simplex iterations. As the matrix representation of the logical vectors is a unit matrix it need not be stored. This part of **A** is best handled in an administrative way.

The data structure for the factored form of the basis or the inverse in product form is usually a compromise between a completely static and a modestly dynamic storage method. Details are given in Chapter 8.

Data structures similar to the ones just discussed can be used if **A** is stored as a collection of sparse row vectors. In some cases both row and columnwise representation is advisable. In other cases a complete columnwise representation and the rowwise access to the positions of the nonzeros (or the other way round) is useful in SSX (e.g., during inversion/factorization). This latter can be achieved if the columnwise data structure is amended by a separate list holding only the indices of the rowwise representation of the matrix. To illustrate it, we add three more arrays to the data structure of Table 5.1. The resulting new structure is shown in Table 5.2 where `rbeg` points to the beginning of the rows, `rlen` is the row length (this is included for greater flexibility) `cind` is the list of column indices of the elements in the rows.

5.4. Linked lists

There are situations where fully dynamic data structures are needed. This dynamism can be achieved by using linked lists. One of the most important purposes of linked lists is to help avoid search. In this section we give an overview of this indispensable data structure to the extent needed for the efficient implementation of the simplex method.

Table 5.2. Matrix **A** stored as a collection of sparse column vectors and its row indices as collection of sparse row vectors.

Subscripts	1	2	3	4	5	6	7	8	9	10
cbeg	1	3	4	5	8					
clen	2	1	1	3	3					
rind	3	1	3	2	4	1	5	3	4	5
val	1.0	-1.0	-1.0	-1.0	4.0	2.0	-3.0	2.0	-1.0	1.0
rbeg	1	3	4	7	9					
rlen	2	1	3	2	2					
cind	1	4	3	1	2	5	4	5	4	5

Elements of a linked list can appear anywhere in an array. Each element has a link (or pointer) associated with it that points to the location of the next element of the list. The link of the last element points to an invalid position that can easily be distinguished from a real link. In terms of the implementation of the simplex method, we can think of pointers as integers giving the index (location) of the next element in the array. There is a special pointer called *header* that points to the location of the first element of the list. Without the header the list is inaccessible. The location of the header must be known in advance.

The elements of the list need not be located in any order and there may be unused positions (gaps) in the array holding the list. An array can accommodate several linked lists that have no common elements. In this case each list has a separate header.

Very often we need to store elements that have several attributes. It can be done by using parallel arrays for each attribute. In the LP practice the attributes require integer, double precision or character arrays though in some cases arrays of bit-streams may be needed. It is to be noted that elements of a linked list can be stored in other ways than in arrays. However, for the implementation of the simplex method arrays seem to be the appropriate way of storage.

As an example, we show how our sample matrix can be stored as a collection of linked column vectors. Each column has a header that points to the first element of the column. Additionally, each nonzero has a pointer pointing to the position of the next nonzero in the linked list. The last element points to null which is meant to be an invalid position. With linked lists, ordering within a column is meaningless. The sequence of elements is determined by the way the pointers have

been set up. In Table 5.3, row **chead** contains the headers of the linked lists of the columns, **rind** is the row index, **val** is the value of the elements and **rlink** is the pointer to the next row index in the column. Note, the elements of some of the columns are not held in consecutive memory locations in this example (typical with linked lists).

Addition and deletion of elements to and from a linked list are the main operations we usually want to do. An example for each can highlight how these operations are performed.

Let the nonzeros of the integer vector $\mathbf{v} = [3, 0, 0, 7, 0, 4, 2, 0, 0]^T$ be stored as a linked list:

Subscripts	1	2	3	4
header	1			
rind	4	6	1	7
val	7	4	3	2
rlink	3	4	2	0

Adding a new element. Assume, we want to change v_9 (the last element of \mathbf{v}) from 0 to 3. In the data structure it means the insertion of an element with row index 9 and value of 3. We simply place this element to the next available place in the array **val** and its row index in the same position in array **rind**. Assuming that position 5 is occupied by something else, we use position 6 for the placement. The links can be adjusted in several ways. The most common practice is to insert the new elements (logically) just after the header. Therefore, we change the header to point to 6 (instead of 1) and set the link of the new element to 1 (where the header originally pointed). The resulting structure is:

Subscripts	1	2	3	4	5	6
header	6					
rind	4	6	1	7	*	9
val	7	4	3	2	*	3
rlink	3	4	2	0	*	1

Table 5.3. Matrix \mathbf{A} of (5.2) stored as a collection of sparse linked column vectors.

Subscripts	1	2	3	4	5	6	7	8	9	10
chead	9	1	10	2	7					
rind	3	1	5	4	3	4	3	5	1	2
val	-1.0	2.0	-3.0	4.0	1.0	-1.0	2.0	1.0	-1.0	-1.0
rlink	0	4	0	3	0	8	6	0	5	0

Deleting an element. Assume the element with row index 1 has to be removed from this enlarged list. It is located in position 3. The link pointing to it (from position 1) changes to the link where link of this element pointed (position 2). The element is not deleted physically but its location is made inaccessible from this linked list. (It is not necessarily a waste of space as the deleted entries can be linked together so that their storage space is available for later insertions. Examples for this situation will be shown in section 9.8.2.) The resulting structure is the following:

Subscripts	1	2	3	4	5	6
header	6					
rind	4	6	*	7	*	9
val	7	4	*	2	*	3
rlink	2	4	*	0	*	1

The modified link in the column of subscript 1 is printed in boldface.

From this example it is clear that deletion is very simple if we can identify the predecessor of the element in question. In the current structure it requires search which can be computationally expensive. With a further extension of the data structure search can be eliminated, as shown next.

Table 5.4. Matrix **A** of (5.2) stored as a collection of sparse doubly linked column vectors.

Subscripts	1	2	3	4	5	6	7	8	9	10
cfhead	9	1	10	2	7					
cbhead	5	1	10	3	8					
rind	3	1	5	4	3	4	3	5	1	2
val	-1.0	2.0	-3.0	4.0	1.0	-1.0	2.0	1.0	-1.0	-1.0
rflink	0	4	0	3	0	8	6	0	5	0
rblink	0	0	4	2	9	7	0	6	0	0

If a matrix represents an LP basis it is typical that new elements have to be inserted into columns (or rows) during factorization/inversion. They can be placed to the end of the array holding the data and adjust the pointers (see ‘adding a new element’). It may happen that an element has to be deleted from the structure. This operation can be done efficiently if not only forward but also backward links are maintained

within each column (or row). In this case it is advisable to have additional headers which point to the last element of each column. The backward pointing linked list has the same properties as the forward list. Table 5.4 shows matrix **A** of (5.2) represented as a set of sparse doubly linked column vectors. Here, we use **cfhead** and **cbhead** to denote the column head of the forward and backward lists, respectively, while **rflink** and **rblink** stand for row forward and backward link, respectively.

The use of linked list in the implementation of the simplex method is widespread. Sparse computing itself comes with the need for administration of the data. The logic of the simplex method requires the maintenance of further administrative information. It is typical that a number of administrative lists are kept as linked lists. Their proper use can contribute a great deal to the efficiency of the solver.

5.5. Implementation of forward/backward linking

In the implementation of the simplex method there are situations where it is important to know, without search, which rows (columns) have some common feature. An example is the set of rows with equal row count (of the number of nonzeros). It plays a vital role in the speed of inversion/factorization (Chapter 8) and in some of the starting procedures (section 9.8).

To stay with the above example, we form sets of rows with equal row count and set up a linked list for each set. There can be as many such sets as the maximum possible row count (m in case of inversion/factorization and \bar{n} during the quoted starting procedures, where \bar{n} now denotes the number of structural variables). Additionally, any (but only one) of these sets can contain all rows, i.e., can have m elements. To reserve an m^2 or $m\bar{n}$ long array to be able to accommodate any extreme case would be prohibitively wasteful. Fortunately, it is true that the total length of the linked lists is never more than m as we have m rows and each of them can belong to at most one set. This gives the idea to place all lists in the same array and let the pointers assist to distinguish them.

Assume there are n (with a general n) possible sets which is the maximum number of nonzero elements in a row and there are m rows to include in one of the lists. At this stage we assume empty rows have been eliminated.

We have found that operations with the linked lists (adding or deleting entries, identifying elements on them) are the simplest if the organization of the linked list is the following. Set up two integer arrays each of size $m + n$, one for the forward links (**flink**) and the other for the backward links (**blink**). Both the headers and the links are placed in these arrays.

The first m positions contain the link of each row pointing to the next row in the same list or back to the header if it is the last element of a list. As the headers are located in positions $m+1, \dots, m+n$ any pointer to a header is an invalid link thus it is suitable to indicate the end of a list.

Headers are initialized to point to themselves both for forward and backward links. In pseudo-code:

```
/* Initialize */
do j = 1,n
    flink(m+j) = blink(m+j) = m+j
enddo
```

Note, this code can slightly be simplified but we keep this form for reasons that will become clear immediately.

Adding a new element to a list can be done without search or logical testing very efficiently. Assume row i has to be added to list j . The header of this list is in position $m+j$. The new element will be added just after the header. The link of the header is changed to point to the new element and the pointer of the new element is set to point to the element where the header pointed just before this insertion. The backward link is also updated in a way which is quite obvious from the pseudo-code:

```
/* Add */
    k = flink(m+j)
    flink(m+j) = i
    flink(i) = k
    blink(k) = i
    blink(i) = m+j
```

Removing an element i from the list can also be done without search. We do not even have to know which list i belongs to. The idea is to set the forward link of the preceding element to the subsequent element on the list. Updating the backward links requires similar action in the reverse order. This can be achieved by the following very simple pseudo-code:

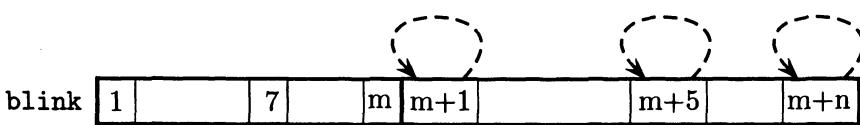
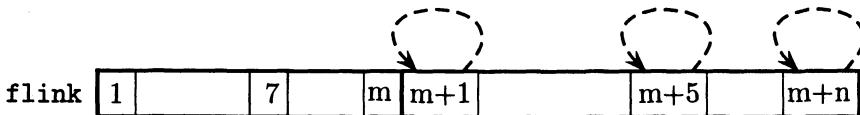
```
/* Remove */
    flink(blink(i)) = flink(i)
    blink(flink(i)) = blink(i)
```

There is no special case, there is no need for testing whether any of the links points to a header.

To illustrate how the above simple procedures work we start with all lists empty. First, row 7 is added to list 5 (which means row 7 has 5 nonzeros). Next row 1 is also added to list 5. After that, row 7 is removed from the list. Finally, row 1 is also removed from the list which leaves the list empty.

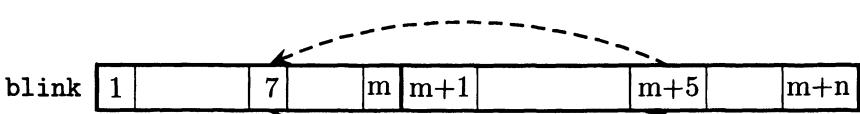
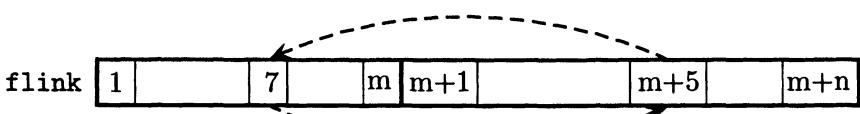
In the graphical representation we use dashed lines for links that have changed since the previous state and solid lines that have remained unchanged.

Initialization. All headers are initialized to point to themselves (represented by self loops).



In the rest of the operations all headers remain self loops except the header of set 5. To simplify the representation, these self loops are not shown in the forthcoming diagrams.

Adding row index 7 to set 5.



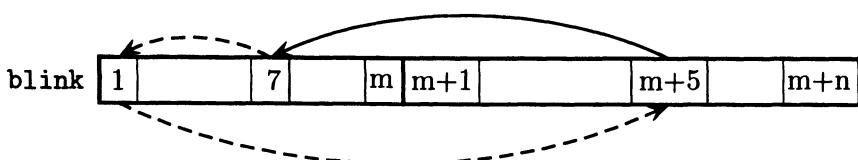
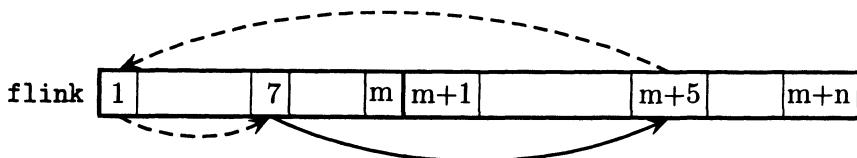
Adding row index 1 to set 5 (row 1 also has 5 nonzeros). Now it is interesting to see how the links change according to the rules of Add.

```

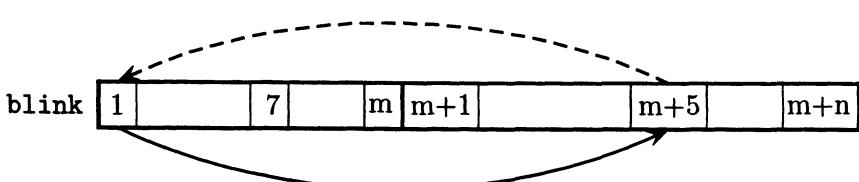
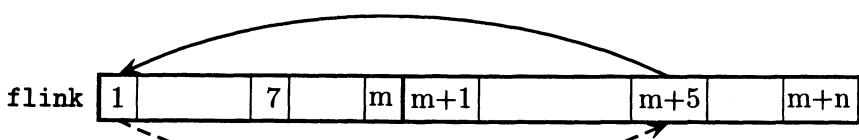
k = flink(m+5) = 7
flink(m+5) = 1
flink(1) = 7
blink(7) = 1
blink(1) = m+5

```

Graphically:

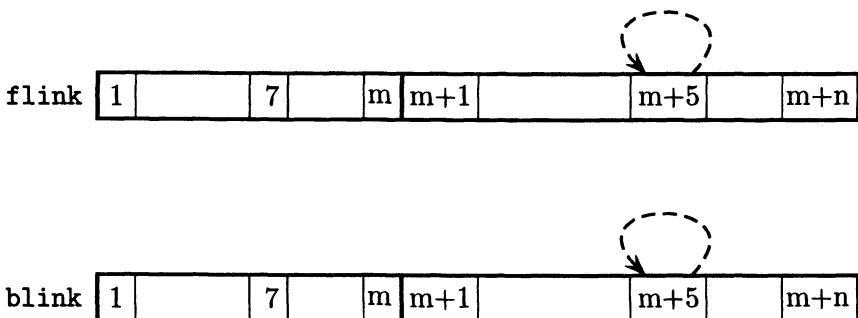


Removing row index 7 from the set which it is now in (by the way, it is set 5 but it really does not matter). Applying the two lines of the Remove code, on the left hand side of the first statement we have `flink(blink(7))` which is `flink(1)` as `blink(7) = 1`. On the right-hand side we have `flink(7)` which is `m+5`. Thus the result is `flink(1) = m+5`. Similar arguments for `blink` give `blink(m+5) = 1`. Graphically:



Removing row index 1 from its list. Now it is particularly interesting to see how the links of the headers reduce to self loops. From the first line of the Remove code we have on the left hand side `flink(blink(1))`

which is $\text{flink}(m+5)$ as $\text{blink}(1) = m+5$. On the right-hand side we have $\text{flink}(1)$ which is $m+5$. The result is then $\text{flink}(m+5) = m+5$. The second line updates the backward link in the following way. On the left hand side $\text{blink}(\text{flink}(1))$ is $\text{blink}(m+5)$. On the right-hand side there is $\text{blink}(1)$ which is $m+5$. Combining the two, we obtain $\text{blink}(m+5) = m+5$. Thus we got back to the self loops for both flink and blink which is the representation of an empty set. Graphically, it is identical with our starting position with all headers initialized to self loops. We show only the reinstated self loops.



It is to be noted that *at no stage* of the example used we *search or conditional statements*. This feature is extremely important in case of very large matrices.

Two further remarks can be made about the use of the above procedures. First, if we know the maximum row count in advance (which can be determined during the input or presolve of the problem or when copying the basis into a work array for inversion or factorization) then the number of headers reduces from n to this number. Thus the size of the link arrays can also be reduced. Often it is advantageous to use smaller arrays to hold the lists. Second, when using the above structure (for whatever purpose), unnecessary visiting of the empty sets can be avoided by maintaining another list that keeps track of the nonempty sets.

5.6. Concluding remarks

This chapter has covered certain aspects of sparse computing, mostly the issues of data structures. Some basic algorithms and procedures have also been presented. They appear sufficient to define the data structures and operations needed for the sparse simplex method.

Readers who want more insight into various further sparsity techniques are advised to turn to some of the well established sources, like Gustavson [Gustavson, 1972], Greenberg [Greenberg, 1978c], Kalan [Kalan, 1971] and Adler, Karmarkar, Resende and Veiga [Adler et al., 1989]. There are two excellent books, Duff, Erisman and Reid [Duff et al., 1986] and George and Liu [George and Liu, 1981]. They present techniques that can widely be applied in various areas, including SSX, with great success. This chapter has particularly benefitted from [Duff et al., 1986].

Finally, it is worth emphasizing again that sparse computing requires good data structures that take into account the types of operations we want to perform on them and also how these operations are organized. These two sides are inseparable and together constitute the basis of the success of the algorithms designed to solve large scale sparse optimization problems.

Chapter 6

PROBLEM DEFINITION

From practical point of view, linear programming problems have two different representations. The final version of a model created by a modeling system or a special purpose program is presented in some computer readable form, usually in a file. It is called the *external representation*. This file is submitted to a solver which first converts it into an *internal representation*.

The most widely used external representation is the MPS (Mathematical Programming System) format which is considered a de facto industry standard. The internal representation is not standardized. In practice, it is different for each implementation. The only common feature of them is that they all take advantage of the sparsity of the matrix of constraints using some combination of techniques outlined in Chapter 5.

In this chapter we discuss the MPS format and give some hints how it can be processed efficiently.

6.1. The MPS format

The MPS format was introduced by the mathematical programming package of IBM in the 1950s. It reflects the state of computer technology at that time when all primary data had to be entered to a computer on punched cards. These cards had 80 characters each of which could represent a letter, a decimal digit or a symbol (from a limited set). Thus, a natural unit of input was an 80 character long record.

To stay within these limitations bad compromises had to be made. As it will be seen the MPS format is rather awkward and lacks any type of flexibility. Since a very large number of important models have been stored in this format, for compatibility (and convenience) reasons it survived all (otherwise rather vague) efforts to replace it with a more

suitable one. As such it is a must for every serious system to be able to input problems in this format. In practice, solvers usually accept at least two external formats, one of them always being MPS. It is not uncommon that the conversion of the MPS file is done by a separate program which creates an intermediate file and the solver uses it as a direct input.

Because of its importance, the description of the MPS format cannot be missed in a book dealing with the computational aspects of solving LP problems. The basic philosophy of it is correct even today, namely, give only the nonzeros of the problem together with their position to reduce the amount of input data.

As it can be expected from the preliminaries, the MPS format is very rigid. It divides up a record into six strictly positioned fields as shown below (location is the character position within the record):

	Field-1	Field-2	Field-3	Field-4	Field-5	Field-6
Location	2-3	5-12	15-22	25-36	40-47	50-61
Contents	Indicator	Name	Name	Value	Name	Value

The file is vertically structured and consists of sections. Each section is introduced by a section identifier record which contains one of the following character strings starting from position 1:

NAME
ROWS
COLUMNS
RHS
RANGES
BOUNDS
ENDATA

The sections must appear in this order. RANGES and BOUNDS are optional.

In the MPS format every entity (constraint, variable, etc.) has a unique identifier (name) consisting of up to 8 characters. The advantages of identifying everything symbolically are obvious. Originally, the MPS format was designed to store a single matrix but several right-hand side, objective, range and bound vectors. They are distinguished by unique names. At solution time the actually used vectors had to be specified. In the absence of the specification the first of each was taken and none of the optional ranges and bounds. The utilization of these possibilities has eroded over the decades and the current practice is to include just one of these vectors which are then automatically selected.

The objective function cannot be given as a separate entity. It is included in the matrix as a non-binding row. The default understanding is that the first non-binding row is considered the objective function. If none is given an error message is triggered.

Each matrix element is identified by a triple in a coordinate system where the coordinates are represented by names like

[column name] [row name] [value]

Similarly, any information relating to a row is identified by the row name. For instance, the structure of defining a b_i element is

[row name] [value]

The same applies to a range value. Obviously, bound entries are identified in a similar structure by column names.

It is to be noted that zero valued entries can also be given, though they most probably will not be stored internally. Such explicit zeros are usually placeholders for modeling purposes.

Comment can be included at any place in any record. A comment is introduced by an * (asterisk) character. Any information, including the asterisk, up to the end of the record is ignored by the input. It is quite typical that comments are included in the MPS file. It is a good practice to give a brief annotation of the problem, date of creation, person responsible, etc. If the problem has already been solved and is meant for other people to test algorithms with the problem then information about the solution can also be included.

The names are expected to start with a non-space character. It is highly recommended to use decimal point in the value fields to avoid an implementation dependent default interpretation of its assumed location when not given.

The individual sections are defined in the following way.

- 1 **NAME.** This section consists of just this record. The string NAME must be in positions 1–4. Additionally, the LP problem can be given a title in positions 15–22. Actual implementations allow the title to be any long up to the end of the record.

This is the first record of the problem definition. Any information before it is ignored by the input.

- 2 **ROWS.** In this section the names of all rows are defined. No other name can appear later in the COLUMNS section. Additionally, the type of the constraints are also defined here. The following notations are used.

Row type	Code and meaning
\leq	L less than or equal constraint
\geq	G greater than or equal constraint
=	E equality constraint
Objective	N objective row
Free	N non-binding constraint

The code can appear in one of positions 2 or 3 (unnecessary freedom!), the row name must be in Field-2.

- 3 COLUMNS. The section identifier must appear in positions 1–7 in a separate record.

This section does two things. First, it defines the names of the columns (variables), second, it defines the coefficients of the matrix including the objective row. As said before, usually the nonzeros are given but some zero placeholders can also appear here. Therefore, the input function must not assume that all given values are nonzero.

The elements of a column can be given in any order (not necessarily in the order of the row names of section ROWS). However, all elements of a column must be given in one group. It is an error to give some elements of a column then some elements of another and again elements of the first.

The data records have the following structure.

Field-2: Column name (maximum 8 characters).

Field-3: Row name (one of those defined in ROWS section).

Field-4: Value of the coefficient.

This structure is repeated for each element of the current column. There is an optional possibility to define two elements in one record. In this case the second [name][value] pair is given in

Field-5: Row name.

Field-6: Value of the coefficient.

The latter can only be used if the first part of the record is also used for defining a matrix element. In this way two elements can be given in a record. The single and double records can be used intermixed. Obviously, a good practice suggests some uniformity.

- 4 RHS. The section identifier must appear in the first three positions of a separate record.

This section gives the values of the right-hand side coefficients. As several right-hand side vectors can be defined here each vector has its own name. The structure of the data records is

Field-2: Name of the RHS vector.

Field-3: Row name.

Field-4: RHS value.

Field-5 and Field-6 can be used the same way as in COLUMNS. The order of giving the RHS coefficients within an RHS vector is insignificant.

Even if all the RHS values are the RHS section identifier record must be present.

- 5 **RANGES**. This is an optional section. If given the string RANGES must be in the first six positions of a separate record.

A range constraint has been defined in Chapter 1 as

$$L_i \leq \sum_{j=1}^n a_j^i x_j \leq U_i,$$

where both L_i and U_i are finite. The range of the constraint is $R_i = U_i - L_i$. One of the values of L_i or U_i is defined in the RHS section which we denote by b_i now. The R_i value of the range is given in the RANGES section. Together with the row type information given in the ROWS section, the L_i and U_i limits are interpreted as follows:

Row type	Sign of R_i	Lower limit L_i	Upper limit U_i
L (\leq)	+ or -	$b_i - R_i $	b_i
G (\geq)	+ or -	b_i	$b_i + R_i $
E (=)	+	b_i	$b_i + R_i $
E (=)	-	$b_i - R_i $	b_i

The data records of this section are defined the same way as those of the RHS section. The only difference is the RANGES vector name field.

Field-2: Name of the RANGES vector.

Field-3: Row name.

Field-4: Range value (R_i).

Field-5 and Field-6 can be used the same way as in COLUMNS. The order of giving the range coefficients within a RANGES vector is insignificant.

6 BOUNDS. This is also an optional section. If given the string BOUNDS must be in the first six positions of a separate record.

The definition of the bound vectors is a bit unusual. With a single bound name we can define a pair of lower and upper bounds. It means that a variable can have more than one entry in this section under the same bound name, for example, when both a finite lower and an upper bound are defined.

In the MPS format there is a lot of default values defined which are in force unless they are redefined. First of all, if the BOUNDS section is missing all structural variables are assumed to be of type-2 (0 lower bound and $+\infty$ upper bound).

The data records of this section have the following structure:

Field-1: Bound type.

Field-2: Name of the BOUNDS vector.

Field-3: Column name.

Field-4: Bound value.

Fields 5 and 6 are not used and must be blank (or comment).

Bound type is a two character specifier:

Specifier	Description	Meaning
L0	Lower bound	$\ell_j \leq x_j (\leq +\infty)$
UP	Upper bound	$(0 \leq) x_j \leq u_j (\leq +\infty)$
FX	Fixed variable	$x_j = \ell_j$ (which is $= u_j$)
FR	Free variable	$-\infty \leq x_j \leq +\infty$
MI	Unbounded below	$-\infty \leq x_j (\leq 0)$
PL	Unbounded above	$(0 \leq) x_j \leq +\infty$

The assumed default bounds are given in parenthesis.

7 ENDDATA. This is a stand alone record which terminates the file of the LP problem. It must always be given in the first six positions of a separate record. It is often subject to misspelling. Note it is written with one 'D'.

There are some variations possible with the MPS format which are not particularly relevant to the simplex method. Interested readers are recommended to turn to [Murtagh, 1981] for more details.

Among the deficiencies of the MPS format the most peculiar is the absence of the indicator of the sense of optimization. Therefore, the solver

must somehow be advised whether a given problem is a maximization or minimization.

As an example, we give the following LP problems in MPS format.

$$\begin{array}{lllll}
 \text{max} & 4.5x_1 & +2.5x_2 & +4x_3 & +4x_4 \\
 \text{s.t.} & x_1 & & +x_3 & +1.5x_4 \leq 40 \\
 20 \leq & & 1.5x_2 & +0.5x_3 & +0.5x_4 \leq 30 \\
 & 2.5x_1 & +2x_2 & +3x_3 & +2x_4 = 95 \\
 & x_1, x_2 \geq 0, -10 \leq x_3 \leq 20, 0 \leq x_4 \leq 25.
 \end{array}$$

It is assumed the names of the rows are Res-1, Res-2, Balance and the names of the variables are Vol--1, Vol--2, Vol--3, Vol--4. The objective function is called OBJ. Other names are defined at the relevant places.

The MPS file is

```

* This a 3 by 4 sample MPS file created 2002.03.15
NAME      Sample Problem
ROWS
N  OBJ          * Objective function
L  Res-1
L  Res-2
E  Balance
COLUMNS
  Vol--1    OBJ        4.5    Res-1        1.0
  Vol--1    Balance   2.5
  Vol--2    OBJ        2.5    Res-2        1.5
  Vol--2    Balance   2.0
  Vol--3    OBJ        4.0    Res-1        1.0
  Vol--3    Res-2     0.5    Res-3        3.0
  Vol--4    OBJ        4.0    Res-1        1.5
  Vol--4    Res-2     0.5    Res-3        2.0
RHS
  RHS-1    Res-1     40.0   Res-2       30.0
  RHS-1    Balance   95.0
RANGES
  RANGE-1  Balance   10.0
BOUNDS
  LO BOUND-1 Vol--3   -10.0
  UP BOUND-1 Vol--3    20.0
  UP BOUND-1 Vol--4    25.0
ENDATA

```

6.2. Processing the MPS format

Converting an MPS file into internal representation is a major effort. A carefully designed and sophisticated implementation can operate several orders of magnitude faster than a simple straightforward one. There

is little information published about the details of a good design. One exception in the open literature is [Nazareth, 1987] where some important details are given and also pointers to further readings are included. The MSc project report of Khaliq [Khaliq, 1997] with most of the important details is not widely known.

In this section we highlight the complications involved and give guidelines how they can be overcome. The key issue is efficiency and reliability. By the latter we mean that the processing does not break down if errors are encountered in the MPS file but they are detected and appropriate messages are provided that can help the modeler to correct them.

The processing is non-numerical. It mostly consists of creating and using look-up tables. In the internal representation of the LP problem variables and constraints are identified by subscripts. Therefore, the names of the MPS format have to be translated to determine the internal equivalent of, for instance,

[column name] [row name] [value]

or

[row name] [value]

which are a_j^i and b_i .

Processing the MPS format is a surprisingly complicated procedure.

The first thing is to be able to identify the section headers. In the **ROWS** section a table has to be set up that gives the correspondence between row names and row indices. During the setup, duplicate row names have to be identified and ruled out. Also the correctness of the row types has to be checked.

The processing of the **COLUMNS** section is the most complex. When a new column name appears it has to be entered into a table and checked to avoid duplicate column names (elements of a column must be given contiguously). Then the row name is to be looked up in the row names table to get its row index. If it is not on the list the row index cannot be determined thus the element is not included in the matrix. A row name cannot be repeated within a column. It must be checked, too. Finally, the given numerical value also must be tested for syntactical correctness and converted into internal representation (usually using some built-in functions of the programming language, like `read`).

The **RHS** and **RANGES** sections are processed similarly. They involve mostly table look-up, generating error messages if relevant and testing the numerical values. In **RANGES** there is a little extra activity for determining the actual limits according to range table.

The **BOUNDS** section involves more work. First, when a column name occurs it has to be verified by the column names table and its index

retrieved. Then there is some logic needed to sort out what the bound code is (also check for correctness), what it implies, whether there are contradictory entries for a column, etc.

At the end of the processing it is necessary to check whether the required sections were present, objective function was defined. If the answer is ‘no’ to either of these questions it is a fatal error because the problem is not defined properly. Some non-fatal errors can also be highlighted by warnings, e.g., ‘no entry in RHS section’ (the section label must be present even in this case).

The efficient operation of these procedures can be ensured by the application of methods of computer science. In this respect the most important basic algorithms are pattern matching, hashing, sorting and binary search. These techniques are discussed in great detail in [Knuth, 1968]. Practical aspects of their use are well presented in several books by Sedgewick, c.f., [Sedgewick, 1990]. The latter book can also be used to implement some critical algorithmic details of the forthcoming versions of the simplex method.

Chapter 7

LP PREPROCESSING

Large scale LP problems are generally not solved in the form they are submitted to an optimization system. There are a number of things that can be done to the original problem to increase the chances of solving it reliably and efficiently. Problems usually can be simplified and their numerical characteristics improved. The activities dealing with these issues are generally called *preprocessing*. The two main actions are *presolve* which attempts simplification and *scaling* which tries to improve the numerical characteristics of the problem. During presolve the problem undergoes equivalent transformations. In Chapter 1, when the LP problem was brought to a computationally more suitable form, we did something that also can be viewed as preprocessing. However, because of its nature it had to be discussed separately.

When a solution is obtained the values of the variables of the original problem have to be restored. This procedure is called *postsolve*. Though it is not a real preprocessing activity this chapter is the best place to discuss it. Therefore, we go through each of the above outlined procedures in some detail that is sufficient for a successful implementation of the most important elements of them. Additionally, we provide references to original research papers for readers interested in more details.

7.1. Presolve

Large scale LP problems are usually generated by some general purpose modeling systems or purpose built programs. As the problem instances are much larger than humans can comprehend it is inevitable that some weaknesses in a model remain overlooked. Examples are the inclusion of empty rows/columns, redundant constraints that unnecessarily increase the size and complexity of the problem. If they are de-

tected and removed a smaller problem can be passed over to the solver. Several other beneficial effects can be achieved by suitably designed and implemented presolve algorithms.

In general, the main objectives of presolve are to

- 1 reduce the size of the problem,
- 2 improve the numerical characteristics, and
- 3 detect infeasibility or unboundedness without solving the problem.

Studying the output of presolve can be instructive. It provides a useful feedback to the modeler and can contribute to improved problem solving already in the modeling phase.

As in many other cases, there is a trade-off between the amount of work and the effectiveness of presolve. The techniques applied consist of a series of simple tests. We can expect more problem reduction if more tests are performed. However, there is no performance guarantee.

In their pioneering work, Brearley, Mitra and Williams [Brearley et al., 1975] suggested the first tests that are capable of (i) detecting and removing redundant constraints, (ii) replacing constraints by simple bounds, (iii) replacing columns by bounds on shadow prices, (iv) fixing variables at their bounds, and (v) removing or tightening redundant bounds. They also noticed the different requirements of the LP and Mixed Integer LP (MILP) presolves.

The importance of the ideas of their work has grown substantially as its benefits have become increasingly evident. Nowadays some version of a presolve is a ‘must’ for an advanced implementation of the simplex method. It is very likely based on [Brearley et al., 1975].

Other important contributions to the ideas of presolve are due to Tomlin and Welch [Tomlin and Welch, 1983a, Tomlin and Welch, 1983b, Tomlin and Welch, 1986], Fourer and Gay [Fourer and Gay, 1993], Andersen and Andersen [Andersen and Andersen, 1995], and Gondzio [Gondzio, 1997]. The latter two also elaborate on the special requirements posed by the interior point methods for linear programming. There is an interesting idea of aggregation that reduces the size of the problem by aggregating constraints. Liesegang gives an in-depth discussion of this topic in [Liesegang, 1980].

To introduce the tests of presolve we use the LP problem with general lower bounds:

$$\min \quad \mathbf{c}^T \mathbf{x} + c_0 \quad (7.1)$$

$$\text{s. t.} \quad \mathbf{A}\mathbf{x} = \mathbf{b} \quad (7.2)$$

$$\ell \leq \mathbf{x} \leq \mathbf{u}, \quad (7.3)$$

allowing some ℓ_j and u_j to be $-\infty$ and $+\infty$, respectively. The c_0 term in the objective function represents its initial value. In general, it is assumed zero and is omitted. However, in the context of presolve it is worth including it explicitly.

The dual of (7.1) – (7.3) is

$$\max \quad \mathbf{b}^T \mathbf{y} + \boldsymbol{\ell}^T \mathbf{d} - \mathbf{u}^T \mathbf{w} \quad (7.4)$$

$$\text{s. t.} \quad \mathbf{A}^T \mathbf{y} + \mathbf{d} - \mathbf{w} = \mathbf{c} \quad (7.5)$$

$$\mathbf{d}, \mathbf{w} \geq \mathbf{0}. \quad (7.6)$$

This form can be obtained by using the developments in section 10.1. All the vectors in (7.1) through (7.6) are of compatible dimension with $\mathbf{A} \in \mathbb{R}^{m \times n}$. It is to be noted that the dual variables, y_i , $i = 1, \dots, m$, are unrestricted in sign.

During presolve some row/column eliminations are identified. They are performed with immediate effect. It means that these entities (rows or columns) and all their nonzero elements are left out from further investigations as if they were not there. Therefore, the running indices in the summations are understood to cover their currently valid range. Also, some updating of the right-hand side or the objective function may be necessary. They are also performed and the b_i or c_j coefficients always refer to their current value.

Several tests are based on the smallest and largest values a constraint can take. To determine them, we define the index sets of the positive and negative coefficients in each row of \mathbf{A} in (7.2).

$$\mathcal{P}_i = \{j : a_j^i > 0\}, \quad (7.7)$$

$$\mathcal{N}_i = \{j : a_j^i < 0\}, \quad (7.8)$$

for the available rows. In this way the smallest value, which will be referred to as *lower limit*, is

$$\underline{b}_i = \sum_{j \in \mathcal{P}_i} a_j^i \ell_j + \sum_{j \in \mathcal{N}_i} a_j^i u_j, \quad (7.9)$$

and the largest value, *upper limit*, is

$$\bar{b}_i = \sum_{j \in \mathcal{P}_i} a_j^i u_j + \sum_{j \in \mathcal{N}_i} a_j^i \ell_j. \quad (7.10)$$

If infinities appear among the ℓ_j and u_j bounds with nonnegative a_j^i the sums they contribute to will be infinite. In such a case it is worth keeping a count of the number of contributing infinities.

7.1.1 Contradicting individual bounds

This is perhaps the simplest possible test. If $\ell_j > u_j$ for any j the problem is obviously infeasible. It is worth testing all $j = 1, \dots, n$ to detect all contradictions of this type.

7.1.2 Empty rows

If the only nonzero in a row i is the ‘1’ of the implied logical variable z_i in (7.2) then all the structural coefficients are zero. Such a row is called an *empty row*. It may be included in the model inadvertently or can be the result of other reductions during presolve. Now we have $z_i = b_i$. Depending on the type of z_i (expressing the type of the constraint) the b_i may be a feasible value for z_i , in which case constraint is redundant and the actual row size of the problem is reduced by one. If b_i is an infeasible value for z_i the constraint is a contradiction and the problem is infeasible.

7.1.3 Empty columns

If column j is empty the corresponding variable does not consume any resources, i.e., does not influence the joint constraints. Depending on the value of c_j the following cases can occur.

- 1 $c_j = 0$ which implies x_j can take any feasible value. Therefore, we can fix it within $\ell_j \leq u_j$ and take it out of the problem. This fixing does not cause any change in the objective function.
- 2 $c_j > 0$ which entails x_j can be fixed at its lower bound ℓ_j as this results in the best contribution to the objective function (minimization). If $\ell_j = -\infty$ the solution is unbounded.
- 3 $c_j < 0$ which implies x_j should be set to its upper bound u_j to contribute the best to the objective. If $u_j = +\infty$ the solution is unbounded.

Fixing a variable with $c_j \neq 0$ causes a change in the objective function. Formally, if we fix $x_j = t$, where t is the chosen value, then in the objective we have $c_0 := c_0 + c_j t$.

7.1.4 Singleton rows

If only one structural coefficient a_j^i is nonzero in row i then it is called a *singleton row*. Such rows can be removed from the problem. If the constraint is an equality the value of x_j is uniquely determined by $x_j = b_i/a_j^i$. If $x_j \notin [\ell_j, u_j]$ then the problem is infeasible. Otherwise, x_j is fixed at this value and is removed from the problem.

Whenever row i is a type-3 (non-binding) constraint x_j can be fixed at any feasible value and removed.

If row i is an inequality constraint then it defines a lower or upper bound for x_j . If a newly evaluated bound is tighter than the currently known one it can be used to replace the old one.

Assuming $a_j^i x_j \leq b_i$ we have the following possibilities.

1 If $a_j^i > 0$ then $x_j \leq \bar{u}_j = b_i/a_j^i$.

2 If $a_j^i < 0$ then $x_j \geq \bar{\ell}_j = b_i/a_j^i$.

In the opposite case, when $a_j^i x_j \geq b_i$,

1 If $a_j^i > 0$ then $x_j \geq \bar{\ell}_j = b_i/a_j^i$.

2 If $a_j^i < 0$ then $x_j \leq \bar{u}_j = b_i/a_j^i$.

If a lower and an upper limit are available for row i such that $\underline{b}_i \leq a_j^i x_j \leq \bar{b}_i$ and any or both of \underline{b}_i and \bar{b}_i are finite then the above relations can be used to evaluate individual bounds on x_j . The new bounds are defined to be the better of the old and the new ones:

$$\ell_j := \max\{\ell_j, \bar{\ell}_j\}, \quad (7.11)$$

$$u_j := \min\{u_j, \bar{u}_j\}. \quad (7.12)$$

7.1.5 Removing fixed variables

A variable becomes fixed if at some stage (or at the beginning) its lower and upper bounds are equal, $\ell_j = x_j = u_j$. Another reason for fixing variables was shown in section 7.1.3. A fixed variable can be removed from the problem but the constant term of the objective and the limits of the joint constraints have to be updated in a way similar to the lower bound translation in (1.13).

To be more specific, let x_j be fixed at a value t , i.e., $x_j = t$. The contribution of x_j to the objective function is $c_j t$ which can be accounted for in the constant term as $c_0 := c_0 + c_j t$. The constant vector $t \mathbf{a}_j$ is taken out of \mathbf{Ax} and is accounted for in the limits of the joint constraints as

$$\bar{\mathbf{b}} := \bar{\mathbf{b}} - t \mathbf{a}_j, \quad (7.13)$$

$$\underline{\mathbf{b}} := \underline{\mathbf{b}} - t \mathbf{a}_j. \quad (7.14)$$

In this way the number of variables is reduced by one with each fixing.

7.1.6 Redundant and forcing constraints

Any \mathbf{x} satisfying (7.2) and (7.3) also satisfies

$$\underline{b}_i \leq \sum_j a_j^i x_j \leq \bar{b}_i, \quad \forall i. \quad (7.15)$$

Assume constraint i was originally a ' \leq ' type constraint:

$$\sum_j a_j^i x_j \leq b_i. \quad (7.16)$$

Comparing b_i with the limits \underline{b}_i and \bar{b}_i enables us to draw some conclusions.

- 1 $b_i < \underline{b}_i$: Constraint (7.16) cannot be satisfied because even the smallest value of $\sum_j a_j^i x_j$ is greater than the upper bound b_i . Thus the problem is infeasible.
- 2 $b_i = \underline{b}_i$: We say that constraint (7.16) is a *forcing* one as it forces all variables involved in the definition of \underline{b}_i to be fixed at those values assumed in the definition. Namely, $x_j = \ell_j$, $j \in \mathcal{P}_i$ and $x_j = u_j$, $j \in \mathcal{N}_i$. All the fixed variables and also constraint i are then removed from the problem. This action reduces the number of constraints by one and the number of variables by $|\mathcal{P}_i| + |\mathcal{N}_i|$.
- 3 $b_i \geq \bar{b}_i$: Constraint i cannot exceed its computed upper limit \bar{b}_i , therefore, it is redundant and can be removed. As a result, the number of constraints is reduced by one.
- 4 $\underline{b}_i < b_i < \bar{b}_i$: Constraint i cannot be eliminated.

If the original constraint is of type ' \geq ' it can be multiplied by -1 and the above procedure can be applied. The other possibility is to evaluate the different cases directly from $\sum_j a_j^i x_j \geq b_i$ using arguments similar to the above ones.

7.1.7 Tightening individual bounds

The computed limits on a constraint can often be used to derive individual bounds on the participating variables. This is quite straightforward if one or both limits are finite. Gondzio [Gondzio, 1997] noticed that it is possible to determine finite bounds if there is only one infinite bound in the equation defining \underline{b}_i or \bar{b}_i . First we discuss the finite case followed by Gondzio's the extension.

For simplicity, we restrict ourselves to the ' \leq ' type constraints (a representative of them is (7.16)) and show the results in this context.

Assume \underline{b}_i is finite and in its definition we have variable x_k such that $k \in \mathcal{P}_i$ which is at its lower bound, $x_k = \ell_k$. If we allow any single variable in \mathcal{P}_i to move away from ℓ_k we obtain

$$\underline{b}_i + a_k^i(x_k - \ell_k) \leq \sum_j a_j^i x_j \leq b_i, \quad k \in \mathcal{P}_i, \quad (7.17)$$

and, in a similar fashion, if x_k , $k \in \mathcal{N}_i$, is at its upper bound,

$$\underline{b}_i + a_k^i(x_k - u_k) \leq \sum_j a_j^i x_j \leq b_i, \quad k \in \mathcal{N}_i. \quad (7.18)$$

From equation (7.17) a lower and from (7.18) an upper bound can be obtained for x_k

$$x_k \leq \bar{u}_k = \ell_k + \frac{b_i - \underline{b}_i}{a_k^i}, \quad k \in \mathcal{P}_i, \quad (7.19)$$

and

$$x_k \geq \bar{\ell}_k = u_k + \frac{b_i - \underline{b}_i}{a_k^i}, \quad k \in \mathcal{N}_i. \quad (7.20)$$

If any of the computed bounds $\bar{\ell}_k$ and \bar{u}_k is tighter than the currently used lower or upper bound the new replaces the old one in the same way as in (7.11) and (7.12). We may also obtain that $\bar{\ell}_k > \bar{u}_k$, in which case the problem is infeasible.

The above calculations are performed for each row which has a zero infinity count. Gondzio proposed an extension of the above for rows that have infinity count equal one.

It is assumed that \underline{b}_i is not finite and there is only one infinite bound, say $\ell_k = -\infty$ for some $k \in \mathcal{P}_i$ or $u_k = +\infty$ for some $k \in \mathcal{N}_i$, in its definition, see (7.9). Now, equations (7.17) and (7.18) can be replaced by

$$a_k^i x_k + \sum_{j \in \mathcal{P}_i \setminus \{k\}} a_j^i \ell_j + \sum_{j \in \mathcal{N}_i} a_j^i u_j \leq \sum_j a_j^i x_j \leq b_i, \quad \text{if } k \in \mathcal{P}_i, \quad (7.21)$$

or

$$a_k^i x_k + \sum_{j \in \mathcal{P}_i} a_j^i \ell_j + \sum_{j \in \mathcal{N}_i \setminus \{k\}} a_j^i u_j \leq \sum_j a_j^i x_j \leq b_i, \quad \text{if } k \in \mathcal{N}_i. \quad (7.22)$$

From (7.21) an upper bound can be derived for x_k

$$x_k = \bar{u}_k \leq \frac{1}{a_k^i} \left(b_i - \sum_{j \in \mathcal{P}_i \setminus \{k\}} a_j^i \ell_j - \sum_{j \in \mathcal{N}_i} a_j^i u_j \right) \quad (7.23)$$

and from (7.22) a lower bound can be obtained for x_k

$$x_k = \bar{\ell}_k \geq \frac{1}{a_k^i} \left(b_i - \sum_{j \in P_i} a_j^i \ell_j - \sum_{j \in N_i \setminus \{k\}} a_j^i u_j \right) \quad (7.24)$$

Again, if any of the computed bounds $\bar{\ell}_k$ and \bar{u}_k is tighter than the currently used lower or upper bound the new replaces the old one in the same way as in (7.11) and (7.12). We may also obtain that $\bar{\ell}_k > \bar{u}_k$, in which case the problem is infeasible.

Finally, we refer to the same two possibilities of handling the ' \geq ' type constraints as in the discussion of the forcing constraints.

It appears that the above extension is particularly useful if the problem contains free variables. Namely, in this case equations (7.17) and (7.18) do not give finite limits while (7.23) and (7.24) can. As a result, individual bounds can be generated for free variables that can lead even to their elimination. If the latter cannot be achieved then it is better to leave them as they were since free variables are very advantageous for the simplex method.

7.1.8 Implied free variables

For a moment, let ℓ_j and u_j denote the original (finite or infinite) individual bounds of x_j , and $\bar{\ell}_j$ and \bar{u}_j the tightest computed ones. We call x_j an *implied free variable* if $[\bar{\ell}_j, \bar{u}_j] \subseteq [\ell_j, u_j]$ holds. In other words, if the row constraints guarantee that x_j stays within tighter limits than its original individual bounds then it can be treated as a free variable which is obviously beneficial.

In the special case when the implied free variable x_j is a column singleton with its nonzero entry in row i ($a_j^i \neq 0$) x_j can be removed from the problem. It is because its removal does not affect any other constraint. At the same time, constraint i can also be removed as it becomes a free row. The actions taken must be recorded because the removed row contains information about the relationship of the removed variable and the other variables that remain in the problem which must be taken into account during postsolve to determine the value of x_j in an optimal solution.

When a row is removed as a result of an implied column singleton we have to ensure that this change is accounted for also in the objective function. It can be seen from the dual constraint (7.5) that a singleton implied free variable x_j uniquely determines the i -th dual variable

$$y_i = \frac{c_j}{a_j^i}. \quad (7.25)$$

This value is always feasible because y_i is a free variable, see (7.5). To take the effect of this fixing into account the objective coefficients have to be modified

$$\bar{\mathbf{c}}^T = \mathbf{c}^T - y_i \mathbf{a}^i, \quad (7.26)$$

where \mathbf{a}^i denotes row i of \mathbf{A} , as usual.

7.1.9 Singleton columns

The case of a free singleton column has been discussed in section 7.1.8. If the variable corresponding to a singleton column has at least one infinite bound we can draw conclusions using the optimality conditions as they impose sign constraints on the components of \mathbf{d} and \mathbf{w} . Namely,

$$\begin{aligned} d_j &= 0 && \text{if } \ell_j = -\infty, \\ w_j &= 0 && \text{if } u_j = +\infty. \end{aligned}$$

It is easy to see that if exactly one of the bounds is finite the corresponding dual constraint in (7.5) becomes an inequality. If $u_j < +\infty$ then $w_j \geq 0$, therefore, by dropping w_j , we obtain

$$\mathbf{a}_j^T \mathbf{y} \leq c_j, \quad (7.27)$$

or if $\ell_j > -\infty$ then $d_j \geq 0$, thus

$$\mathbf{a}_j^T \mathbf{y} \geq c_j. \quad (7.28)$$

If \mathbf{a}_j is a column singleton with a_j^i being the nonzero entry (7.27) and (7.28) are reduced to $a_j^i y_i \leq c_j$ or $a_j^i y_i \geq c_j$, respectively. In this way we can derive a bound for the y_i dual variable. Let $\gamma = c_j/a_j^i$. The following table shows the bounds generated by this column singleton

	$a_j^i > 0$	$a_j^i < 0$
$a_j^i y_i \leq c_j$	$y_i \leq \gamma$	$y_i \geq \gamma$
$a_j^i y_i \geq c_j$	$y_i \geq \gamma$	$y_i \leq \gamma$

7.1.10 Dominated variables

Using dual information there is a chance to draw further useful conclusions on the variables and constraints of the LP problem.

Let the bounds on the dual variables y_i be denoted by p_i and q_i such that

$$p_i \leq y_i \leq q_i, \quad i = 1, \dots, m. \quad (7.29)$$

The index sets of the positive and negative entries in a column are defined as

$$\mathcal{P}'_j = \{i : a_j^i > 0\}, \quad (7.30)$$

$$\mathcal{N}'_j = \{i : a_j^i < 0\}, \quad (7.31)$$

for $j = 1, \dots, n$. In this way the smallest value a dual constraint can take, which will be referred to as lower limit, is

$$\underline{c}_j = \sum_{i \in \mathcal{P}'_j} a_j^i p_i + \sum_{i \in \mathcal{N}'_j} a_j^i q_i, \quad (7.32)$$

and the largest value, upper limit, is

$$\bar{c}_j = \sum_{i \in \mathcal{P}'_j} a_j^i q_i + \sum_{i \in \mathcal{N}'_j} a_j^i p_i. \quad (7.33)$$

If infinities appear among the p_i and q_i bounds with nonnegative a_j^i the sums they contribute to will be infinite. It is worth keeping a count of the number of contributing infinities.

Any dual feasible solution \mathbf{y} satisfies

$$\underline{c}_j \leq \sum_i a_j^i y_i \leq \bar{c}_j, \quad \forall j. \quad (7.34)$$

For simplicity, we assume $u_j = +\infty$, which means the corresponding dual constraint is $\mathbf{a}_j^T \mathbf{y} \leq c_j$ as shown in (7.28). Analyzing (7.34), the following cases can be distinguished and conclusions drawn.

- 1 $c_j < \underline{c}_j$: Dual constraint (7.28) cannot be satisfied, therefore the problem is dual infeasible.
- 2 $c_j > \bar{c}_j$: In this case d_j must be strictly positive to satisfy the j -th component of (7.5). It means x_j is *dominated*, it can be fixed at its finite lower bound, $x_j = \ell_j$ and eliminated from the problem. Obviously, if $\ell_j = -\infty$ the problem is dual unbounded.
- 3 $c_j = \bar{c}_j$: Dual constraint (7.28) is always satisfied with $d_j \geq 0$. The corresponding variable x_j is said to be *weakly dominated*. Under some further conditions (see [Gondzio, 1997]) a weakly dominated variable can also be fixed at its lower bound.
- 4 $\underline{c}_j \leq c_j < \bar{c}_j$: Variable x_j cannot be eliminated from the problem.

A similar analysis can be performed if we assume $\ell_j = -\infty$ in which case the relevant dual constraint, in inequality form, is (7.27).

7.1.11 Reducing the sparsity of A

As the advantages of sparsity of the constraint matrix A are overwhelming it is a good idea to try to increase it (i.e., decrease density). If it cannot be done logically anymore with the tests described so far we may turn to numerical elimination. Gondzio in [Gondzio, 1997] outlined an idea which is simple enough to implement but can also be effective in reducing the number of nonzeros. It is a heuristic algorithm with some minimum performance guarantee. It is based on the analysis of the sparsity pattern of A .

Any equality type constraint can be used to add a scalar multiple of it to any other constraint. This row will be referred to as pivot row. If row i is chosen for this purpose we select another row, say k , whose sparsity pattern is a superset of that of row i . In this case, the sparsity pattern of

$$\bar{\mathbf{a}}^k = \mathbf{a}^k + \gamma \mathbf{a}^i \quad (7.35)$$

is the same as \mathbf{a}^k . If $\bar{\mathbf{a}}^k$ replaces \mathbf{a}^k in the LP problem the corresponding RHS element also changes: $\bar{b}_k = b_k + \gamma b_i$. Obviously, the problem with the new row k is equivalent with the original one.

By a suitable choice of γ in (7.35) we can achieve that at least one nonzero in \mathbf{a}^k is eliminated. With some luck we can experience the elimination of some other nonzero elements. However, the guarantee is just one element per superset row. The beauty of this procedure is that it never produces fill-in and does not require additional storage.

While the idea is simple its efficient implementation is less trivial. The main difficulty lies in the search for rows with superset sparsity to a chosen row i . The choice of the pivot row can be critical. To enhance the chances of finding supersets it is advisable to start with rows having a small number of nonzeros.

The computational effort of identifying the supersets to row i can be reduced in the following way. Take the shortest column that intersects row i at a nonzero. Denote its index by j . Examine only those rows that have a nonzero in column j . If a row is not a superset of row i it usually becomes evident after checking only the first two or three elements. A row can also be rejected if it is shorter than row i . Figure 7.1 shows some typical situations that can occur.

For the efficient implementation of this procedure, a linked list of equality type rows is set up. They are the candidate pivot rows. The list is ordered by increasing number of nonzeros in the the rows. First, the shortest row is selected and the procedure is performed. Each processed pivot row is removed from the list and any equality type row in which a cancellation occurred reenters the list. When no superset can be found to

a	x		x		x		x	↔ pivot row
			x		x		x	
b	⊗	x	x	x	⊗	x	⊗	↔ superset
c	x	x	x		x	x	x	x
				x		x		x
d	x	x		x		x		
e	x	x	x			x	x	x

Figure 7.1. Identifying supersets to row a (pivot row). Column 1 is the shortest intersecting column and it is used to find supersets. Row b qualifies, row c is not considered because of the zero in column 1, row d fails as it is shorter than row a, and row e does not qualify as it fails to match the second nonzero of row a. One of positions \otimes in row b can be eliminated.

any of the candidates (i.e., when the list becomes empty) the procedure stops.

7.1.12 The algorithm

The tests described in the previous sections can be combined to make a presolve algorithm. It is not necessary to include all discussed tests. On the other hand some more can be added from the references given in section 7.1. The more tests are included the more reduction can be expected. However, in this case computational effort and time will also increase.

After a setup when counts are initialized and sets are determined, the matrix is scanned several times. In every pass empty rows and columns are identified first and the appropriate actions taken. Then, each column is checked whether the corresponding variable is redundant or unbounded or provides a bound on the dual variable. Redundant variables are fixed at the determined level and removed from the problem. The remaining problem is modified to account for the changes. During a pass, individual bounds are calculated. If they are tighter they replace the old ones. At the end of the pass the new bounds are used to recompute the row limits and identify redundant or infeasible rows. Singleton rows are removed and the corresponding variables are fixed or individual bounds are generated. Also, implied free variables are identified.

Because of the iterative nature of the presolve algorithm a change in one pass may lead to further changes in the next. In general, if no reduction is achieved in two successive passes the algorithm terminates. However, theoretically it is possible that an infinite sequence of small

improvements in the bounds are generated, as pointed out by Fourer and Gay [Fourer and Gay, 1993] on a contrived example, in which case the algorithm cannot stop. Therefore, it is worth setting an upper limit on the number of passes.

7.1.13 Implementation

There are two main issues that require special attention for an efficient and effective presolve algorithm: the use of appropriate data structures and the proper handling of numerical operations.

Regarding data structures, certain techniques were already recommended in section 7.1.11. Some additional points are as follows. It is beneficial to keep \mathbf{A} in both column- and rowwise forms. This double storage scheme can also be used during the simplex algorithm to enhance efficiency. Details of it are discussed in section 9.4.1. To avoid too much search linked lists can be set up that can be used in several tests and also in the housekeeping of the entire presolve. An important question is how the elimination of a row is performed. A good practice in the columnwise representation is the use of pointers to the start of the columns and keeping the lengths of the columns in terms of the number of nonzeros. In this case, if an element is to be deleted, it can be swapped with the last nonzero in the column and the length decreased by one. This way the ordering of the elements in the column changes but it is not a problem as no ordering is assumed by any of the algorithmic units of the simplex method.

It may come as a surprise but presolve is susceptible to numerical problems. The original matrix elements are stored in double precision and they remain unchanged during presolve. Their accuracy is determined during input conversion. The matrix elements may change if the algorithm for making \mathbf{A} sparser is activated. The new bounds and limits are always obtained by calculations and may carry round-off and cancellation errors. Therefore, in evaluating, for instance, the forcing constraints, infeasibility or unboundedness of the problem, special care must be exercised. Fourer and Gay [Fourer and Gay, 1993] noticed that the use of directed, rounding when available with a processor, can increase the robustness of the procedure. When this facility is not available a tolerance should be used in comparisons of computed quantities. It may also help if the positive and negative terms of dot products are accumulated separately and added at the end. This can decrease the chances of cancellation error. Even more sophisticated techniques can be used if numerical problems persist.

Unfortunately, it is not easy to recognize that numerical errors caused problems (wrong conclusions) in presolve. In particular, wrong indica-

tion of infeasibility or unboundedness are not easy to verify without actually solving the un-presolved problem. Therefore, it is important that presolve is coded for robust operation. In this respect the proper use of tolerances is inevitable. They have to be used in comparisons for equality and also for inequality of computed quantities, like instead of $a = b$ we check whether $|a - b| < \epsilon$, where $\epsilon > 0$ is a tolerance in the magnitude of $10^{-12} - 10^{-10}$.

7.2. Scaling

Scaling in the LP context is a linear transformation of the matrix of constraints whereby each row and each column is multiplied by a real number called *scale factor*. It entails the adjustment of the other components of the LP problem, namely, right-hand side, objective function and bounds. Formally, scaling is equivalent to pre- and postmultiplying \mathbf{A} by diagonal matrices, such that

$$\mathbf{R}\mathbf{AC}\bar{\mathbf{x}} = \mathbf{R}\mathbf{b}, \quad (7.36)$$

where $\mathbf{R} = \text{diag}\langle R_1, \dots, R_m \rangle$, $\mathbf{C} = \text{diag}\langle C_1, \dots, C_n \rangle$, and $\mathbf{C}\bar{\mathbf{x}} = \mathbf{x}$. From the latter

$$\bar{\mathbf{x}} = \mathbf{C}^{-1}\mathbf{x}. \quad (7.37)$$

Obviously, the finite individual bounds of \bar{x}_j are also transformed to u_j/C_j and, similarly, the explicit lower bounds, if given, to ℓ_j/C_j . The objective row is not scaled rowwise (implicitly, the factor is one). However the computed column scale factors can be applied to it. Thus, the objective coefficient of \bar{x}_j becomes $c_j C_j$ (a bit of a notational hiccup). It ensures that

$$\bar{c}_j \bar{x}_j = c_j C_j x_j \frac{1}{C_j} = c_j x_j,$$

i.e., the objective values of the scaled and unscaled problems are the same. It can be useful when the iterations are monitored. The unit column vectors of logical variables are not scaled. Therefore, they are added to the LP problem after scaling.

The purpose of scaling is to improve the numerical characteristics of an LP problem and thus enable a safe solution. The importance of scaling in computational linear algebra has long been known. Unfortunately, its role in linear programming is not that well understood. There are examples where scaling makes it possible to solve an otherwise unsolvable problem but there are other examples where just the opposite occurs: a problem cannot be solved when scaled but becomes easily solvable without scaling. The main difference between traditional linear algebra and LP is that the former works with the matrix of a fixed set of equations

while in LP the set of (basic) equations keeps changing and each of them may require a different scaling. However, what can be done cheaply is a one-off scaling of A which then applies to all bases selected by the simplex method.

Good modeling practice can contribute to the good numerical characteristics of a problem. Namely, the proper choice of units of measurements of the variables usually leads to a balanced magnitude of the matrix elements which is believed, and in many cases proved, to be beneficial. Tomlin [Tomlin, 1975] has conducted a systematic review of scaling and attempted to make some recommendations. Though his results are inconclusive there are some interesting observations in his paper. We will use some of his arguments in the sequel.

It is quite difficult to characterize well scaled matrices. The opposite is somewhat easier. We say that a matrix is *badly scaled* if the magnitudes of the nonzero elements are vastly different. Practice has shown that such matrices are quite difficult to process (solve systems of equations with them, factorize, invert). As a contraposition, we can say (or believe) that a matrix is *well scaled* if the magnitudes of its nonzero elements are close to each other. Obviously, both definitions lack any quantitative measure. The tendency is that if scaling reduces the spread of the magnitudes then the problem is easier to solve. There are several counterexamples, though. One of them is a matrix with elements $a_j^i = 1 \pm \varepsilon_j^i$, where $\varepsilon_j^i > 0$ is a small random perturbation, not greater than 10^{-8} .

For measuring the spread of the magnitudes of the nonzero elements of an LP matrix, there are two methods in circulation. Fulkerson and Wolfe [Fulkerson and Wolfe, 1962] and also Orchard-Hays [Orchard-Hays, 1968] have recommended

$$\frac{\max |a_j^i|}{\min |a_j^i|}, \quad \text{over all } a_j^i \neq 0, \quad (7.38)$$

and say a matrix is well scaled if this quantity is not larger than a threshold value of about $10^6 - 10^8$. The other measure is due to Hamming [Hamming, 1971] and Curtis and Reid [Curtis and Reid, 1972]

$$\sum_{a_j^i \neq 0} (\log |a_j^i|)^2. \quad (7.39)$$

They say a matrix is well scaled if this number has an acceptable value. The smaller this value the better the scaling of the matrix is. It is not possible to give some numerical estimates for this type of acceptability because (7.39) is an absolute measure which keeps growing by the

number of nonzero entries, ν_A . A ‘standard deviation like’ measure can be obtained for the average magnitude if the square root of (7.39) is normalized by ν_A . The main argument in favor of using (7.39) and not (7.38) is that (7.39) is less sensitive to a single extreme value than (7.38).

While the main purpose of scaling is to increase the accuracy of processing the matrices, scaling itself can be a source of generating small errors. If the scale factors are not the powers of the base of the floating point number representation of the computer then every time a multiplication or division is performed there is a chance of introducing a round-off error. It can be avoided if the scale factors are restricted to be some powers of the base. This leaves the mantissa, and thus the accuracy, unchanged. Assuming binary representation, the nonzero coefficients of the matrix can be written in the normalized floating point form of

$$a_j^i = f_j^i 2^{e_j^i}, \quad \text{where } \frac{1}{2} \leq f_j^i < 1. \quad (7.40)$$

The scale factors will be powers of 2,

$$R_i = 2^{\rho_i}, \quad \text{and} \quad C_j = 2^{\gamma_j}, \quad (7.41)$$

where e_j^i , ρ_i and γ_j are integers. If we scale the matrix as in (7.36) then the exponents of the scaled elements, denoted by \bar{e}_j^i , become

$$\bar{e}_j^i = e_j^i + \rho_i + \gamma_j. \quad (7.42)$$

It is easy to see (by using base 2 logarithm) that minimizing (7.39) is equivalent to

$$\min_{\substack{\rho_i, \gamma_j \\ a_j^i \neq 0}} g(\rho, \gamma) = \sum (e_j^i + \rho_i + \gamma_j)^2. \quad (7.43)$$

This is a least squares problem that has to be solved for ρ_i , $i = 1, \dots, m$ and γ_j , $j = 1, \dots, n$. For the first glance it seems to be a huge task but Curtis and Reid [Curtis and Reid, 1972] have shown a very efficient way of solving it approximately. Exact solution is not really needed since the final values have to be rounded to the nearest integers anyhow to obtain round-off free scale factors.

Before turning our attention to solving (7.43) we briefly overview other, less sophisticated, scaling methods. They are all heuristics aiming at reducing the spread of the magnitudes of the nonzeros. As such, there is no performance guarantee and, indeed, in some cases they (or some of them) may lead just to the opposite.

Equilibration. Equilibration seems to be the best established standard method. Each row is scaled to make the largest absolute element

equal 1. It is achieved by multiplying the row with the reciprocal of the largest absolute element (so the row scale factors are these reciprocals). It is followed by a similar scaling for the columns (i.e., the column scale factors are the reciprocals of the absolute column maximum values). As a result, the largest absolute element in each row and each column will be 1. This method can generate small round-off errors.

Geometric mean. For each row the

$$(\max_j \{|a_j^i|\} \times \min_j \{|a_j^i|\})^{\frac{1}{2}}$$

quantity is computed and its reciprocal (or its nearest power of 2) is used to multiply the nonzeros of the row. A similar action is taken for each column.

Arithmetic mean. Each row is multiplied by the reciprocal of the arithmetic mean of the nonzero elements in the row (or the nearest power of 2), followed by a similar action for the columns.

In practice, these simple methods are used in combination, one after the other. If round-off is not an issue the last method in a combination is always equilibration. The reason for it is that it gives some sense to setting some absolute tolerances in the LP problem. As such, a typical combination is geometric mean followed by equilibration.

An interesting dynamic scaling technique has been proposed by Benichou at al., [Benichou et al., 1977]. Their method is the choice in the IBM MPSX system. First, they apply geometric scaling one to four times. The variance of the nonzero elements of the scaled matrix, defined as

$$\frac{1}{\nu_A} \left(\sum (a_j^i)^2 - \frac{\left(\sum |a_j^i| \right)^2}{\nu_A} \right),$$

where a_j^i denotes the elements of the scaled matrix and ν_A is the number of nonzeros in \mathbf{A} . It is evaluated after every pass. If it falls below a given value (they recommend 10) geometric scaling is stopped. This procedure is followed by equilibration. Though this scaling can introduce small rounding errors it has become quite popular because it turned out to be quite helpful in numerically difficult cases.

Now we turn to solving (7.43) and present a special conjugate gradient method recommended by Curtis and Reid [Curtis and Reid, 1972]. We can refer to their procedure as a sort of ‘optimal scaling’ as it aims at finding the best scaling factors using (7.43) as the objective function. Setting the partial derivatives of g with respect to ρ_i , $i = 1, \dots, m$ and

γ_j , $j = 1, \dots, n$ equal to zero we obtain, after rearrangement, a set of linear equations in $m + n$ variables in the following way

$$\sum_{j:a_j^i \neq 0} (\rho_i + \gamma_j) = - \sum_{j:a_j^i \neq 0} e_j^i, \quad i = 1, \dots, m, \quad (7.44)$$

$$\sum_{i:a_j^i \neq 0} (\rho_i + \gamma_j) = - \sum_{i:a_j^i \neq 0} e_j^i, \quad j = 1, \dots, n. \quad (7.45)$$

Let r_i and c_j denote the nonzero count of row i and column j , respectively. (This is an exceptional use of c_j as it is usually reserved to denote the objective coefficients of the LP problems.) Then, (7.44) and (7.45) can be rewritten as

$$r_i \rho_i + \sum_{j:a_j^i \neq 0} \gamma_j = - \sum_{j:a_j^i \neq 0} e_j^i, \quad i = 1, \dots, m, \quad (7.46)$$

$$\sum_{i:a_j^i \neq 0} \rho_i + c_j \gamma_j = - \sum_{i:a_j^i \neq 0} e_j^i, \quad j = 1, \dots, n. \quad (7.47)$$

If we define the boolean matrix \mathbf{Z} to represent the nonzero pattern of \mathbf{A} , i.e., $z_j^i = 1$, if $a_j^i \neq 0$ and $z_j^i = 0$ otherwise, then (7.46) and (7.47) can be written as

$$\mathbf{M}\boldsymbol{\rho} + \mathbf{Z}\boldsymbol{\gamma} = \mathbf{s} \quad (7.48)$$

$$\mathbf{Z}^T\boldsymbol{\rho} + \mathbf{N}\boldsymbol{\gamma} = \mathbf{t}, \quad (7.49)$$

where $\mathbf{M} = \text{diag}\langle r_1, \dots, r_m \rangle$, $\mathbf{N} = \text{diag}\langle c_1, \dots, c_n \rangle$, and $s_i = - \sum_j e_j^i$ and $t_j = - \sum_i e_j^i$ are the components of \mathbf{s} and \mathbf{t} , respectively. In matrix form,

$$\begin{bmatrix} \mathbf{M} & \mathbf{Z} \\ \mathbf{Z}^T & \mathbf{N} \end{bmatrix} \begin{bmatrix} \boldsymbol{\rho} \\ \boldsymbol{\gamma} \end{bmatrix} = \begin{bmatrix} \mathbf{s} \\ \mathbf{t} \end{bmatrix}. \quad (7.50)$$

The matrix on the left hand side of (7.50) is singular because the sum of the m equations in (7.48) is equal to the sum of the n equations in (7.49). Additionally, this matrix is symmetric, positive semidefinite, and possesses Young's 'property A'. Therefore, a special algorithm of [Curtis and Reid, 1972] can be used that reduces both the computational effort and storage requirements. It is an iterative procedure with excellent convergence properties.

In the setup phase of the algorithm an initial solution is chosen. It is composed of the trivial solution to (7.48), $\boldsymbol{\rho}_0 = \mathbf{M}^{-1}\mathbf{s}$ and $\boldsymbol{\gamma} = \mathbf{0}$. The residual with this solution is

$$\begin{bmatrix} \mathbf{0} \\ \boldsymbol{\delta}_0 \end{bmatrix}, \quad \text{where } \boldsymbol{\delta}_0 = \mathbf{t} - \mathbf{Z}^T\mathbf{M}^{-1}\mathbf{s}.$$

Later residuals can be determined recursively by setting $h_{-1} = 0$, $\delta_{-1} = \mathbf{0}$, $q_0 = 1$, $s_0 = \delta_0^T \mathbf{N}^{-1} \delta_0$ and using the following equations in a bit liberal way as δ_{j+1} will be n dimensional if j is even and m dimensional if j is odd:

$$\delta_{j+1} = \begin{cases} -\frac{1}{q_j} (\mathbf{Z} \mathbf{N}^{-1} \delta_j + h_{j-1} \delta_{j-1}) & \text{if } j \text{ is even,} \\ -\frac{1}{q_j} (\mathbf{Z}^T \mathbf{M}^{-1} \delta_j + h_{j-1} \delta_{j-1}) & \text{if } j \text{ is odd,} \end{cases} \quad (7.51)$$

furthermore,

$$s_{j+1} = \begin{cases} \delta_{j+1}^T \mathbf{N}^{-1} \delta_{j+1} & \text{if } j \text{ is odd,} \\ \delta_{j+1}^T \mathbf{M}^{-1} \delta_{j+1} & \text{if } j \text{ is even,} \end{cases} \quad (7.52)$$

and

$$h_j = q_j \frac{s_{j+1}}{s_j} \quad (7.53)$$

$$q_{j+1} = 1 - h_j, \quad (7.54)$$

for $j = 0, 1, 2, \dots$. It is obvious from (7.51) that the dimension of subsequent δ residuals is m and n , alternating. Consequently, the residuals of (7.50) will be

$$\begin{bmatrix} \mathbf{0} \\ \delta_j \end{bmatrix}$$

if j is even or

$$\begin{bmatrix} \delta_j \\ \mathbf{0} \end{bmatrix}$$

if j is odd.

There is a possibility to save computational work by updating only one of the vectors ρ_j and γ_j at each iteration. To determine γ_{2k+2} the following recursion can be used

$$\gamma_{2k+2} = \gamma_{2k} + \frac{1}{q_{2k} q_{2k+1}} [\mathbf{N}^{-1} \delta_{2k} + h_{2k-1} h_{2k-2} (\gamma_{2k} - \gamma_{2k-2})], \quad (7.55)$$

for $k = 0, 1, \dots$, with $h_{-2} = 0$, $\gamma_{-2} = \mathbf{0}$, and for ρ_{2k+3} we can use

$$\rho_{2k+3} = \rho_{2k+1} + \frac{1}{q_{2k+1} q_{2k+2}} [\mathbf{M}^{-1} \delta_{2k} + h_{2k-1} h_{2m} (\rho_{2k+1} - \rho_{2k-1})], \quad (7.56)$$

for $k = 0, 1, \dots$, with $\rho_{-1} = \rho_1 = \rho_0$. To synchronize the two updating formulas at the termination of the iterations, one or the other of the following formulas has to be used

$$\gamma_{2k+1} = \gamma_{2k} + \frac{1}{q_{2k}} [\mathbf{N}^{-1} \delta_{2k} + h_{2k-1} h_{2k-2} (\gamma_{2k} - \gamma_{2k-2})] \quad (7.57)$$

and

$$\rho_{2k+2} = \rho_{2k+1} + \frac{1}{q_{2k+1}} [\mathbf{M}^{-1} \rho_{2k+1} h_{2k-1} h_{2k} (\rho_{2k+1} - \rho_{2k-1})]. \quad (7.58)$$

As a stopping criterion

$$s_{j+1} \leq 0.01\nu_A \quad (7.59)$$

is used where ν_A is the number of nonzeros in \mathbf{A} .

The above procedure may look complicated. However, it can be implemented quite easily. In practice it converges in no more than $8 - 10$ iterations regardless of the size of the problem and the time taken is just a small fraction of the ensuing simplex iterations. Very often it is sufficient to make $4 - 5$ iterations. When an approximate solution to (7.43) is obtained, all ρ_i and γ_j values are rounded to the nearest integer giving $\bar{\rho}_i$ and $\bar{\gamma}_j$. The scale factors are determined from $R_i = 2^{\bar{\rho}_i}$ and $C_j = 2^{\bar{\gamma}_j}$.

It is important to note that scaling can cause an adverse effect. For instance, there are models with a large number of ± 1 entries which is generally very advantageous for the solution algorithm. It is a source of spontaneous cancellations during factorization and updating which is a welcome event. Such a problem certainly does not benefit from scaling since many of the ± 1 values may undergo scaling and become distinct. Therefore, in an advanced implementation there must be provision for, maybe different versions of, scaling and also the possibility of solving the problem unscaled.

Scaling does not require sophisticated data structures but the row- and columnwise access to \mathbf{A} is important. It can be achieved by storing \mathbf{A} in both ways. This storage is very useful in several other algorithmic units, therefore, it is highly recommended if memory allows.

7.3. Postsolve

As a result of preprocessing (including problem reformulation discussed in Chapter 1) the problem actually solved is equivalent to the original one but not identical with it. Therefore, the solution obtained has to be transformed back to present it in terms of the original variables and constraints. This activity is called postsolve. During postsolve all the changes have to be undone using the values of the variables provided by the solver.

The basic requirement of postsolve is to have all changes of preprocessing properly recorded. These changes are best kept in a stack type data structure where always the most recently made action is on the top. The rationale behind it is that the changes have to be undone in the reverse order. If we perform the relevant ‘undo’ operation with the information on the top of the stack, it can be deleted and the next becomes the top as long as the stack is not empty.

The usual order of the actions during preprocessing is

- (i) problem reformulation, see Chapter 1,
- (ii) presolve, section 7.1
- (iii) scaling, section 7.2.

As they involve different types of operations it is practical to set up three separate stacks. They can be implemented in many different ways. While it is not a difficult exercise, it would be difficult to say which is the best solution. We leave this issue open for the reader.

7.3.1 Unscaling

The first thing to do with the solution is to express it in terms of the unscaled problem (assuming the problem was scaled). It requires undoing the effects of scaling. Whatever combination of scaling procedures was used, ultimately it was expressed as a pre- and postmultiplication of \mathbf{A} by diagonal matrices as shown in (7.36):

$$\mathbf{R}\mathbf{AC}\bar{\mathbf{x}} = \mathbf{R}\mathbf{b},$$

where $\mathbf{R} = \text{diag}\langle R_1, \dots, R_m \rangle$, $\mathbf{C} = \text{diag}\langle C_1, \dots, C_n \rangle$, and $\mathbf{C}\bar{\mathbf{x}} = \mathbf{x}$, with $\bar{\mathbf{x}}$ representing the solution of the scaled problem. Therefore, the unscaled values of the structural variables are obtained if their computed values are multiplied by the corresponding column scale factors.

The values of the logical variables are determined by the solver to equate the left hand side with the scaled right-hand side. To restore their unscaled value they have to be divided by the corresponding row scale factors.

If the objective row was scaled the reduced costs (dual logicals) also have to be unscaled by dividing them by the column scale factors.

Finally, the shadow prices (the reduced costs of the logical variables which are the dual variables) have to be multiplied by the scale factors of the rows they belong.

7.3.2 Undo presolve

Perhaps this is the most complicated part of reversing the actions of preprocessing. Even this can be done relatively easily if the events of presolve have been properly recorded. Gondzio in [Gondzio, 1997] calls it *presolve history list* and represents every modification to the variables (both primal and dual) by a triple consisting of two integers and a double precision real number (i_1, i_2, a) . The integers identify the variable(s) involved and the type of the action while the real number represents the numerical information, like value of fixing or change of the bound. For example, a lower bound correction can be recorded as $(i_1 = j, i_2 = 0, a = (\Delta)\ell_j)$. In this case the first integer stores the index of the variable, the second integer denotes the type of the operation. The real number can represent the change in the bound or the new bound itself. The latter is somewhat more practical. As another example, if an implied free variable is identified it can be stored as $(i_1 = j, i_2 = -i, a = y_i)$, where y_i comes from equation (7.25). Now the sign of the second integer defines the type of the action.

Note, fixing a variable at a certain value entails a translation of the right-hand side, see equations (1.13), (7.11) and (7.12).

Most of the steps of presolve are trivial to undo as the inverse operations are uniquely defined. This applies equally to modifications of the primal and dual variables. The important thing is to follow the strict reverse order of actions.

There is one case that deserves special attention. Namely, if the reduction of sparsity is included in presolve (section 7.1.11) then regaining the dual information requires a little care. In one step of the reduction two rows are involved in a Gaussian elimination type operation. It can also be represented by a triple $(i_1 = i, i_2 = k, a = \gamma)$, where the symbols refer to equation (7.35). If several such steps have been performed their product can be represented by a nonsingular matrix \mathbf{M} (which need not be formed explicitly). As a result of this operation we have $\bar{\mathbf{A}} = \mathbf{M}\mathbf{A}$. Therefore, the dual variables at an optimal solution satisfy

$$\mathbf{A}^T \mathbf{M}^T \bar{\mathbf{y}}^* + \mathbf{d}^* - \mathbf{w}^* = \bar{\mathbf{c}},$$

where superscript * refers to the optimal values. From this expression the dual solution of the original problem can be obtained as

$$\mathbf{y}^* = \mathbf{M}^T \bar{\mathbf{y}}^*.$$

It is our belief that the presolve techniques discussed in section 7.1 are sufficient for most cases when the problem is being prepared for the simplex method. Readers, interested in more details about presolve and

postsolve are advised to study [Andersen and Andersen, 1995], [Gondzio, 1997] where some more tests are described together with their reverse operations.

7.3.3 Undo reformulation

Development of computational forms # and #2 required very simple operations, see Chapter 1. They included multiplying a row or a column by -1 , moving some finite bounds of variables to zero and creating range type constraints. They entailed possible transformations in the individual bounds of the variables, the objective coefficients and also the right-hand side elements. It was emphasized that these changes had to be recorded to enable the restoration of the original problem after solution.

If the operations are recorded on a stack they can be rewound in a reverse order. Their simplicity does not make it necessary to give an itemized description of them. Readers are referred to Chapter 1 from where all necessary steps can easily be reconstructed and implemented.

Chapter 8

BASIS INVERSE, FACTORIZATION

Throughout the development of the simplex method in Chapter 2 the basis and its inverse have played a particularly important role. At certain stages of the solution algorithm the inverse of the basis was needed to update row and/or column vectors as required by the simplex method. The typical operations were

$$\alpha = B^{-1}a \quad (8.1)$$

and

$$\pi^T = h^T B^{-1} \quad (8.2)$$

(not exactly with these vectors). Thus there is a need to maintain the inverse of the basis in some form to be able to perform these operations.

(8.1) and (8.2) can be viewed slightly differently. In order to obtain α of (8.1) we need solve

$$B\alpha = a \quad (8.3)$$

for α , while π of (8.2) can be obtained if we solve a linear system of equations with the transpose of B

$$B^T \pi = h. \quad (8.4)$$

In the formal description of the simplex method the formulas with the inverse were more natural. However, in computational practice it is not necessarily the best way. In this chapter we discuss both possibilities and present the appropriate algorithms that enable the use of either version.

Historically, the simplex method became a serious computational tool when the revised simplex method was introduced. It was defined in terms of the inverse of the basis. The next important step was the introduction

of the *Product Form of the Inverse* (PFI) by Dantzig and Orchard-Hays [Dantzig and Orchard-Hays, 1954] which enabled the solution of large LP problems. The inherent flexibility available with the product form gave space to different versions that tried to utilize sparsity and achieve better numerical accuracy. The development and use of PFI are presented in section 8.1 of this chapter.

In case of numerically more difficult problems the PFI was not always successful. Sometimes the basis was found singular or the representation of the inverse was numerically not accurate enough which prevented the progress of the simplex method. In other cases the sparsity of the product form was not satisfactory.

Using the (8.3) and (8.4) view of the problem a different approach is possible. If an LU factorization of \mathbf{B} is available then the quoted systems can be solved by back- and forward substitutions. The LU method has better numerical properties and leads to at least as sparse representation as the PFI but involves more sophistication. However, its excellent overall performance makes it utterly important to make it available in a reliable and fast implementation of the simplex method. The details of the LU approach are presented in section 8.2.

The sparse versions of both the PFI and LU methods heavily rely on row and column counts of nonzeros. They are traditionally denoted by r_i (for row i) and c_j (for column j). In this chapter we follow this tradition and temporarily forget about the other use of c_j of being an objective coefficient in the LP problem.

Whichever method is used in a simplex solver, the product form or the LU factors grow in size, operations with them become slow and loaded with increasing numerical inaccuracies. Therefore, the representation has to be recomputed regularly or occasionally if some extraordinary event occurs. This operation is called *reinversion* (in case of PFI) or *refactorization* (for LU). There are some desirable features that reinversion and refactorization are expected to satisfy:

- Speedy operation to cause minimum delay in optimization.
- Generation of sparse PFI or LU to speed up the subsequent operations in the simplex.
- High numerical accuracy to restore the accuracy of the LP solution and that of the iterations.

8.1. Product form of the inverse

As hinted above, first the PFI is discussed in some detail. It is easier to implement than the LU factorization. At the same time, it still can

perform very well for quite sizeable, but no too difficult, LP problems. As a first effort of implementing the simplex method it can be a reasonable candidate to represent the inverse of the basis.

8.1.1 General form

The foundations of the PFI have already been laid down in section 2.1.3. The principle idea is the following. If we know the \mathbf{B}^{-1} inverse of a basis \mathbf{B} then the inverse of any of its neighboring bases, say $\bar{\mathbf{B}}$ can be determined by premultiplying \mathbf{B}^{-1} by an elementary transformation matrix \mathbf{E} (also called ETM) as $\bar{\mathbf{B}}^{-1} = \mathbf{E}\mathbf{B}^{-1}$. For details, see section 2.1.3. This idea can be used to develop the product form of the inverse.

Assume, the columns in the basis are denoted by \mathbf{b}_i , $i = 1, \dots, m$, so that $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_m]$. Inversion in product form can be described very concisely.

- 1 Start with a matrix with known inverse. The unit matrix \mathbf{I} is perfect for this purpose as $\mathbf{I}^{-1} = \mathbf{I}$.
- 2 Replace columns of \mathbf{I} by \mathbf{b}_i , $i = 1, \dots, m$ (assuming it can be done in this order). This corresponds to taking a neighboring basis in each step. In this way the update formula of the basis inverse can be used always with the new ETM.

After m steps we have: $\mathbf{B}^{-1} = \mathbf{E}_m \mathbf{E}_{m-1} \dots \mathbf{E}_1$, where

$$\mathbf{E}_i = [\mathbf{e}_1, \dots, \mathbf{e}_{i-1}, \boldsymbol{\eta}_i, \mathbf{e}_{i+1}, \dots, \mathbf{e}_m]$$

with

$$\boldsymbol{\eta}_i = \left[-\frac{v_i^1}{v_i^i}, \dots, -\frac{v_i^{i-1}}{v_i^i}, \frac{1}{v_i^i}, -\frac{v_i^{i+1}}{v_i^i}, \dots, -\frac{v_i^m}{v_i^i} \right]^T \quad (8.5)$$

$$\mathbf{v}_i = \mathbf{B}_{i-1}^{-1} \mathbf{b}_i, \quad (8.6)$$

$$\begin{aligned} \mathbf{B}_i^{-1} &= \mathbf{E}_i \mathbf{E}_{i-1} \dots \mathbf{E}_1, \\ &= \mathbf{E}_i \mathbf{B}_{i-1}^{-1}, \quad i = 1, \dots, m, \quad \text{and } \mathbf{B}_0^{-1} = \mathbf{I}. \end{aligned}$$

The pivot element in step i is v_i^i .

At the end of the procedure, we have m $\boldsymbol{\eta}$ vectors (each of m components) plus the position of each of them in the unit matrix. Therefore, no more than $m \times m$ elements and m pointers have to be stored. In case of large scale problems (which feature sparsity) the storage requirements are drastically reduced as the $\boldsymbol{\eta}$ vectors (the nontrivial columns of \mathbf{E}_i -s) can also be very sparse.

In practice the operations of this procedure are performed in a slightly different order. When a new ETM is formed, all the remaining columns are updated with it. In this way if in the next step a column is chosen to replace the next unit vector it is directly available to produce a new ETM without the $\mathbf{v}_i = \mathbf{B}_{i-1}^{-1} \mathbf{b}_i$ step.

In the above description it was assumed that in step i \mathbf{b}_i can replace \mathbf{e}_i in the basis, i.e., the pivot element v_i^i is nonzero. If this assumption fails we can try elements in other updated columns v_j^i , $j = i + 1, \dots, m$. If any of them is nonzero its column can be permuted in position i and the element used as the next pivot. If all of them are zero then \mathbf{e}_i cannot be replaced, therefore, \mathbf{B} is singular.

In case of LP bases the creation of the PFI can further be simplified. If there is a unit vector of a logical variable in the basis it generates an identity ETM, i.e., $\mathbf{E}_i = \mathbf{I}$ that can be omitted from the product form. If there are many logicals in the basis PFI may contain considerably fewer factors than m while an explicit inverse \mathbf{B}^{-1} is always of $m \times m$. In practice it is not uncommon that 5–25% of the vectors in the basis are unit columns.

8.1.2 Sparsity exploiting form

While the inverse of a nonsingular matrix (like a simplex basis) is uniquely defined, the product form of the inverse is not. There is a limited freedom in deciding which basic column replaces which unit column of the original identity matrix. This freedom can be used to create sparser and numerically more stable representations of the inverse.

If the basis matrix \mathbf{B} is lower triangular (every element above the main diagonal is zero) the inversion in PFI can be performed with very little computational work. To verify this statement, recall (3.2). It says $\mathbf{E}\mathbf{a} = \mathbf{a}$ if $a_p = 0$, where p is the position of the nontrivial column of \mathbf{E} . It immediately entails that, if \mathbf{B} is lower triangular, no updating is needed in (8.6) because \mathbf{b}_i has a zero entry in all previous pivot positions. Therefore, at stage i we have $\mathbf{v}_i = \mathbf{b}_i$. Even the little computing of forming an $\boldsymbol{\eta}_i$ vector from an original \mathbf{b}_i is vastly reduced as the columns of \mathbf{B} are generally very sparse.

8.1.2.1 The Markowitz merit number

Fill-in can occur when we update a column with previously generated ETMs. It is very desirable to find such pivots that minimize the fill-in. To achieve global minimum a dynamic programming problem should be solved which is well beyond a sensible effort. There are different local strategies that try to minimize the fill-in just for one step ahead. Even this goal is quite ambitious in terms of the effort of keeping track of

existing nonzeros and the changes of the nonzero pattern after a chosen pivot.

Recalling how fill-in is created (Figure 3.3) the following observation can be made. Assume we have updated nonzero counts for the active part of the matrix, i.e., for nonzeros in the rows and columns that have not been pivotal yet. If we choose to pivot on position (i, j) the pivot column has c_j nonzeros (including the pivot element) and the pivot row has r_i nonzeros (together with the pivot). Clearly, in terms of nonzeros, row i intersects $r_i - 1$ columns. In the worst case every off-pivot element of column j causes a fill-in which amounts to $(r_i - 1)(c_j - 1)$ new nonzeros. This is an upper bound for the number of fill-ins and is called the *Markowitz merit number*. It can be computed for each nonzero potential pivot position in the unused rows and columns (the *active submatrix*):

$$m_j^i = (r_i - 1)(c_j - 1). \quad (8.7)$$

It is important to stress that the Markowitz number is an upper bound. An obvious goal can be to find (i, j) for which m_j^i is minimal to avoid the generation of many new nonzeros. In practice, this strategy performs very well and keeps the fill-in low.

From (8.7) it is clear that $m_j^i = 0$ if either $r_i = 1$ (row singleton) or $c_j = 1$ (column singleton). It is easy to see that there is no fill-in in either case.

To find a position with minimum Markowitz number is rather demanding. The sparsity pattern of the active submatrix is needed that can be accessed row- and columnwise. The columnwise pattern helps identify which rows will change with a given pivot column. Additionally, we need to access the numerical values of the nonzero positions to be able to ensure sufficiently large pivot elements. In most practical cases the search for the minimum Markowitz number is cleverly limited to a relatively small number of positions. This gives rise to different methods. Some of them are discussed in the subsequent sections.

8.1.2.2 Numerical considerations

As long as we use original elements (i.e., ones that have not been transformed by an ETM) the numerical values can be viewed as accurate. Therefore any of them can be chosen as pivot without too much precaution.

If the active submatrix has undergone some updating then the newly chosen pivot elements may be computed numbers potentially carrying numerical errors. Detailed error analysis shows (c.f., [Wilkinson, 1965] or [Duff et al., 1986]) that in general it is good to choose large elements for pivoting because it reduces the growth of the magnitude of

the transformed elements. As a consequence, it is not sufficient to look for a position with a minimum or small merit number. It is equally important to find numerically acceptable pivot elements.

A practical compromise between the two requirements is the use of *threshold pivoting*. Broadly speaking it selects a pivot with maximum magnitude from among a set of candidates with good merit number. More specifically, if the selected pivot column is j then the pivot element v_j^i is chosen to satisfy

$$|v_j^i| \geq u \max\{|v_k^i|\}, \quad (8.8)$$

where k is restricted to the rows of the active submatrix and u is the *threshold parameter*. Similarly, if the pivot row is selected first than the pivot element v_j^i is chosen to satisfy

$$|v_j^i| \geq u \max\{|v_k^i|\}, \quad (8.9)$$

where, again, k runs over the column indices of the active submatrix. Meaningful values for u are in the range $0 < u \leq 1$. In practice $u = 0.1$ is understood to be a good default value which generally leads to sparse and acceptably stable PFIs. For many problems even $u = 0.01$ is a good choice.

8.1.2.3 Triangularization method

In practice the large scale LP bases nearly always have a large hidden triangular part. By ‘hidden’ we mean that the presence of the triangular part is not obvious by inspection. However, by appropriate row and column permutations the basis matrix can be brought into a form that exhibits triangularity. *Triangularization* of the basis is the key to the efficiency of PFI.

There are different triangular forms that are desirable for specific versions of PFI. Their common feature is the search for a permutation that gives the largest possible triangular part. The distinctive features reflect the different ways the non-triangular part is treated.

The logical vectors in a basis need not be considered as they all define the trivial unit factor. Therefore, the logical vectors can be marked as being in their home position. It means that their rows and columns are ignored during PFI as if they were not there. The remaining part of the matrix can be permuted into a lower block triangular form as shown in Figure 8.1. Here, the RR and CC blocks are real lower triangular matrices. If the MM block is empty the entire matrix is lower triangular.

Once \mathbf{B} is brought into the form of Figure 8.1 the pivot order is the following. Take the vectors in the R section in the order they are

	<i>R</i>	<i>M</i>	<i>C</i>	
<i>R</i>	x			1
	x x			2
		x		3
	x x	x		4
		x x x	x	5
<i>M</i>	x	x x	x x	6
	x x	x x		7
		x	x x	8
<i>C</i>	x	x x	x x	9
	1 2 3 4 5 6 7 8 9			

Figure 8.1. Lower block triangular form of a sparse matrix with two true triangular blocks RR and CC . Nonzeros are denoted by ‘x’ and zeros by empty cells.

listed and pivot down the diagonal in the RR block. This operation can be performed without any updating of type (8.6) hence no fill-in is generated. Next, choose the columns in the M section pivoting in the MM block. This operation will create fill-in in the MM and CM blocks (but nowhere else). Special techniques can be used to reduce the fill-in and maintain numerical stability. When this block is done, pivot down the diagonal of the CC block which is, again, free from updating.

It is to be noted that permutations are not performed physically. They are just recorded by the pivot sequence as explained shortly.

The important question is how the form of Figure 8.1 can be obtained. There are some possibilities but all of them rely on counts of nonzeros in the rows and columns of the basis matrix \mathbf{B} .

The RR block can easily be identified if we notice that the top left element in position $(1, 1)$ of Figure 8.1 is characterized by being the only nonzero in row 1. In other words, if we find a nonzero count $r_i = 1$ we have identified a potential element for the top left corner. If the nonzero appears in column j then the first pivot position will be (i, j) which is equivalent to permuting row i in the first row and column j in the first column. An η vector is created from this column by pivoting on position i to obtain \mathbf{E}_1 . Pivot position i is recorded for \mathbf{E}_1 and j is entered in the basis heading as $k_1 = j$. Row i is marked unavailable for further pivoting and column j is deleted from the list of variables to enter. The row counts of rows that have nonzeros in this column are

decremented by 1 which is easily done by scanning the indices of the nonzeros of column j . This procedure is repeated for the remaining part of the matrix as long as there is an updated row count equal to 1.

When block RR is completely identified we move over to finding block CC . The bottom right element is such that there are no other nonzeros in its column. Therefore, the following can be done. Look for a column count $c_j = 1$. If one is found, record the row index of the nonzero element, say i . The last position to pivot on will be (i, j) . Delete column j from the list, mark row i unavailable for further pivoting and decrement the counts of those columns that had a nonzero in row i . Record the pivot position and enter j in the basis heading as $k_m = j$. Do not create an η vector yet. Repeat this procedure for the remaining part of the matrix moving backward in the basis heading as long as there is an updated column count with the value of 1. The result of this procedure is a list of the pivot positions that define block CC .

As nonzeros in blocks RR and CC are not updated every pivot is an original (untransformed) element of the basis, therefore they can be viewed as accurate values.

What is left after the above procedures is section M . Columns in M can be included in the PFI by pivoting on positions of the MM block. Very often MM is called the *kernel*. It is characterized by having at least two nonzeros in each row and column. The pivot order in the kernel is crucial. This is where fill-in occurs which we want to minimize. Additionally, it is very likely that the pivot elements are transformed values that can carry some computational error. Therefore, numerical issues also need to be given due consideration. How the kernel is processed makes the difference in the versions of the PFI. Whatever method is used, after the identification of the pivot position the η vector and the corresponding E are formed and the remaining columns in the M section are updated by E . Fill-in occurs as described in section 3.4 of Chapter 3. The pivotal row is marked unavailable, the column is deleted and the row and column counts are updated. Also, the index of the column is entered in the position of the replaced unit vector (moving forward). This procedure is repeated until all columns in section M are processed.

The final stage of the PFI is the creation of the η vectors of the CC block in the reverse order of their identification, i.e, we pivot down the diagonal of CC . In this stage there is no update, hence no fill-in.

If at any stage a row count r_i is zero no candidate that can enter the basis in position i . It indicates that the basis is singular. To repair singularity and let the simplex method proceed, we can keep e_i , the logical vector of row i , in the basis. It does not generate an η . Unfortunately,

as a result, there may be a deterioration in the objective value or the feasibility of the LP solution.

If a column count becomes zero the column cannot enter the basis which, again, indicates singularity of the matrix. In this case there may be ambiguity which row cannot be pivotal. Therefore, the decision must be deferred regarding which unit vector should take the role of this discarded column. At the end of the processing of the kernel all unused pivot positions are assigned the corresponding unit vectors together with their logical variables.

From among the several possibilities of processing the kernel we describe one method and give reference to the others.

8.1.3 Implementing the PFI

The efficient operation of the procedure for obtaining the product form of the inverse heavily depends on the use of appropriate data structures that help exploit sparsity.

The input to the inversion is the list of basic variables that can be found in the basis heading, $\mathcal{B} = \{k_1, \dots, k_m\}$. The logical variables in the basis may not be in their home position. However, it does not matter as PFI starts with the all-logical matrix and replaces the unit columns one by one. Obviously, the basic logical variables are not replaced. They remain in their home position and their rows and columns are marked unavailable for the rest of the procedure. In PFI we deal with the available columns and rows. At the beginning it is the active submatrix.

During initialization, the following counts and lists are set up:

- Row counts r_i for the available rows.
- Column counts c_j for the available columns.
- Row lists of the column indices of nonzeros in each available row. The length of the i -th row list is r_i . The method of storage is described in Chapter 5. It is worth leaving some extra space between rows to be able to accommodate a certain number of, say 5–10, fill-ins. It will reduce the need of row compression.
- Column lists of the row indices and elements of the nonzero positions in each column. These lists can just be copied from the columnwise representation of the original matrix \mathbf{A} . The length of the j -th list is c_j . Method of storage is as above, again, with some space between columns to allow fill-in without the need of creating a fresh copy of the updated column.

When the row counts have been determined and the maximum row length rm is known we can set up a doubly linked list of rows with equal length. We show the pseudo-code of this procedure following the footsteps of section 5.5. The length of the link arrays is $m+rm$. Assume the links have been initialized to self loops and r_i is represented by $r(i)$. If a row is unavailable (position of a logical variable) it is marked by $r_i = 0$. ($r_i = -(i + m)$ can also be used to indicate that the i -th logical variable is basic in position i .)

```
/* Add each row to the list of its row count */
do i=1,m
  if (r(i) > 0)      /* row is not excluded    */
    j = r(i)          /* # of nonzeros in row i */
    k = flink(j+m)
    flink(j+m) = i   /* add row i to list j    */
    flink(i) = k
    blink(k) = i
    blink(i) = j+m
  endif
enddo
```

After this setup, the rows with row count equals 1 (section R) can be visited and processed as long as $flink(m+1)$ is not greater than m .

During processing section R , if a row with $r_i = 1$ is selected in the above way (which is always the one to which the header of list 1 points) the row list is looked at and column j is identified. This column is processed (η vector created, j is entered in position i of the new basis heading: $k_i = j$). Next, the administration needs to be updated in the following way: (i) decrease the counts of all rows that have a nonzero in this column, (ii) remove these rows from their current list and (iii) add them to the list of rows with one less row count (if this number is positive). Obviously, these three steps are performed one after the other for a row being considered when going through the list of column j . Finally, as column j is deleted, its index has to be removed from all rows where it appears. While these rows can directly be accessed they have to be searched to find column index j and delete it. This search does not seem to be heavy, on the average just few elements of the few rows are involved. To avoid this search a much more sophisticated data structure would be needed.

The C section can be processed in an analogous way. It can be even faster than R as generally few columns are left after R has been identified.

The main effort in PFI is the processing of the M section. It has been observed that if some (very few) columns are temporarily ignored then further triangularity can be found in MM . This is a sort of a forced triangularization [Dobosy, 1970, Hellerman and Rarick, 1971, Hellerman and Rarick, 1972]. The main question here is for how long should the processing of these columns be delayed, i.e., when to enter them. If they are deferred to the end then, very likely, many of the ETMs generated so far in MM will update them. It can result in relatively dense and not too accurate columns. Additionally, at the end, there are very few available pivot positions remaining. They can be too small and also inaccurate which can lead to a numerically unstable inverse. In the worst case the consequences can be catastrophic. Despite the possibility of this gloomy scenario the PFI, on the average, performs well. It is particularly true if the ideas introduced in [Duff et al., 1986] are used. Experience shows that the number of nonzeros in the η vectors of the ETMs is generally not more than the double of the nonzeros in \mathbf{B} .

Processing the MM block can be attempted on the basis of the Markowitz merit numbers using threshold pivoting. To avoid the costly computing of m_j^i for all nonzero positions we can approximate it roughly by assuming that small merit numbers occur in columns with small nonzero count. In this way it is enough to check very few (2–4) columns and in each of them apply the (8.8) criterion and select a pivot with the smallest m_j^i from among the positions that satisfy (8.8). It is followed by forming the new ETM and updating the remaining columns in the M section. Only those columns have to be updated that have a nonzero in the pivot row. These columns can easily be identified by the row list of nonzero indices. The η vectors extend to all position (with a theoretical maximum length of m). In practice they are very sparse and generate few fill-ins in the remaining columns. Therefore, if some storage space is available at the end of each column the fill-ins can go there without the need to create a fresh copy of the updated column. Therefore column compression probably can be avoided. The indices of the fill-ins need to be added to the row and column list of indices of nonzeros.

8.1.4 Operations with the PFI

In practice we do not need the inverse in explicit form. The resulting vectors of (8.1) and (8.2) can be obtained without actually multiplying the \mathbf{E} factors before an operation. This section gives the details how it can be done.

In the forthcoming discussion it is assumed that there are k factors in the PFI, i.e.,

$$\mathbf{B}^{-1} = \mathbf{E}_k \cdots \mathbf{E}_1. \quad (8.10)$$

8.1.4.1 FTRAN

With PFI operation $\alpha = \mathbf{B}^{-1}\mathbf{a}$ of (8.1) is defined as

$$\alpha = \mathbf{B}^{-1}\mathbf{a} = \mathbf{E}_k \cdots \mathbf{E}_1 \mathbf{a}.$$

This computation is best performed in a recursive way:

$$\begin{aligned}\alpha_0 &= \mathbf{a} \\ \alpha_i &= \mathbf{E}_i \alpha_{i-1}, \quad i = 1, \dots, k \\ \alpha &= \alpha_k.\end{aligned}\tag{8.11}$$

This operation is widely known as *FTRAN*, an acronym for *Forward TRAnsformation*. The name indicates that the ETMs are taken in a forward order of their creation.

FTRAN is a well behaved operation which can exploit sparsity. We have already shown in (3.2) that an ETM can be skipped if the vector to be transformed has a zero element in the pivot position of the ETM. As \mathbf{a} is assumed to be very sparse there is a good chance that just a few of the ETMs will actually be needed in (8.11). This results in execution speed and better numerical accuracy.

8.1.4.2 BTRAN

Using PFI, operation $\pi^T = \mathbf{h}^T \mathbf{B}^{-1}$ of (8.2) is defined as

$$\pi^T = \mathbf{h}^T \mathbf{B}^{-1} = \mathbf{h}^T \mathbf{E}_k \cdots \mathbf{E}_1.$$

The following recursion can be used to perform this operation:

$$\begin{aligned}\mathbf{h}_0^T &= \mathbf{h}^T \\ \mathbf{h}_i^T &= \mathbf{h}_{i-1}^T \mathbf{E}_{k-i+1}, \quad i = 1, \dots, k \\ \pi^T &= \mathbf{h}_k^T\end{aligned}\tag{8.12}$$

The widely used acronym of this operation is *BTRAN* for *Backward TRAnsformation* to reflect the order the ETMs are used.

To further investigate one step of (8.12) we simplify notation to avoid the use of too many sub and superscripts. The computational step is

$$\begin{aligned}\gamma^T &= \mathbf{h}^T \mathbf{E} \\ &= [h_1, \dots, \boxed{h_p}, \dots, h_m] \begin{bmatrix} 1 & \eta^1 & & \\ & \ddots & \vdots & \\ & & \eta^p & \\ & & \vdots & \ddots \\ & & \eta^m & 1 \end{bmatrix}, \\ &= \left[h_1, \dots, \sum_{i=1}^m h_i \eta^i, \dots, h_m \right]\end{aligned}$$

i.e., $\gamma_i = h_i$ for all $i \neq p$ and $\gamma_p = \sum_{i=1}^m h_i \eta^i$. It means that only one component of the vector changes and its new value is defined by the dot product $\mathbf{h}^T \boldsymbol{\eta}$. Dot products are known to be able to show extreme numerical behavior. As a consequence, the resulting vector of the (8.12) operation can be quite inaccurate. It can lead to all sorts of problems in the simplex method including a wrong indication of improving variables. BTRAN is the source of most numerical troubles in SSX if they occur.

Unfortunately, a simplification, similar to FTRAN, is not available for BTRAN if the columnwise data structure is used. However, one interesting remark still can be made about BTRAN. As in each step of BTRAN just one component of the transformed vector can change, the fill-in is limited to at most one per step because, if $h_p \neq 0$ there is effectively no fill-in.

8.1.4.3 Updating the PFI

Updating the PFI is extremely simple. It is one of its most attractive features. The change of the basis inverse as we move over to a neighboring basis has been discussed in Chapter 2. In particular, equation (2.25) gives a formula which says that the inverse of the new basis can be obtained by premultiplying the old inverse by a new ETM. This ETM is formed from the updated incoming column as shown in (2.22) or (8.5) and (8.6). If the new ETM is denoted by \mathbf{E}_{k+1} then, from (8.10), we have

$$\bar{\mathbf{B}}^{-1} = \mathbf{E}_{k+1} \mathbf{B}^{-1} = \mathbf{E}_{k+1} \mathbf{E}_k \cdots \mathbf{E}_1. \quad (8.13)$$

It means that we only have to create the $\boldsymbol{\eta}$ vector and store it. It is formed from the updated incoming column which is available as it was used in the ratio test. No further operation is needed for the update.

8.1.4.4 Reinversion

(8.13) indicates that the product form keeps increasing in size. The PFI preserves the ETM even of those columns that have left the basis. The growing size means slower FTRAN and BTRAN and also deteriorating numerical accuracy. After a while it is better to spend some time with recomputing the PFI from the current basic columns. It results in a shorter and more accurate representation of \mathbf{B}^{-1} . This action is often referred to as the *reinversion* of the basis.

When to reinvert? If it is done too frequently then the progress of the simplex method is held up, perhaps unnecessarily. If it is done too rarely iteration speed will slow down and numerical problems are more likely to appear. The best one can do is to make some compromise between the extremes. Interestingly, reinversion has to be performed rather fre-

quently and the frequency is nearly independent of the size of the basis (in terms of m). There are recommendations that reinversion has to be done after every 30–50 iterations with the PFI. Another possibility is ‘clock control’. It says when the iteration speed drops below a critical value a reinversion must be performed. The critical value is determined on the basis of the statistics of the first few fixed reinversions. Later it is monitored and updated. Specific details are not given here as the fixed frequency method performs quite well in practice. It is usually a run parameter that can be changed by the user.

8.2. LU factorization

The introduction of the elimination form of the inverse (EFI) and its application to the simplex method is due to Markowitz [Markowitz, 1957]. The elimination leads to the LU decomposition of the basis in the following form

$$\mathbf{B} = \mathbf{L}\mathbf{U}$$

where \mathbf{L} is a lower and \mathbf{U} is an upper triangular matrix. In this way, equation (8.3) can be written as

$$\mathbf{L}\mathbf{U}\boldsymbol{\alpha} = \mathbf{a}, \quad (8.14)$$

which has to be solved for $\boldsymbol{\alpha}$. Substituting $\boldsymbol{\gamma}$ for $\mathbf{U}\boldsymbol{\alpha}$, $\boldsymbol{\alpha}$ can be determined in two steps. First, solve

$$\mathbf{L}\boldsymbol{\gamma} = \mathbf{a} \quad (8.15)$$

for $\boldsymbol{\gamma}$ then solve

$$\mathbf{U}\boldsymbol{\alpha} = \boldsymbol{\gamma} \quad (8.16)$$

for $\boldsymbol{\alpha}$. (8.15) can be solved by forward substitution while (8.16) by back-substitution. Obviously, the resulting $\boldsymbol{\alpha}$ is the solution of (8.14).

Similarly, equation (8.4) can be written as

$$\mathbf{U}^T\mathbf{L}^T\boldsymbol{\pi} = \mathbf{h}, \quad (8.17)$$

which is to be solved for $\boldsymbol{\pi}$. Here we substitute $\boldsymbol{\gamma}$ for $\mathbf{L}^T\boldsymbol{\pi}$ then (8.17) can be solved in the following two steps. First, solve

$$\mathbf{U}^T\boldsymbol{\gamma} = \mathbf{h} \quad (8.18)$$

for $\boldsymbol{\gamma}$ then solve

$$\mathbf{L}^T\boldsymbol{\pi} = \boldsymbol{\gamma} \quad (8.19)$$

for $\boldsymbol{\pi}$. As \mathbf{U}^T is a lower and \mathbf{L}^T is an upper triangular matrix (8.18) can be solved by forward substitution and (8.19) by back-substitution.

If we have a $\mathbf{B} = \mathbf{LU}$ factorization the inverse of \mathbf{B} is given by $\mathbf{B}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}$.

It is very easy to determine the inverse of a lower triangular matrix in product form, see section 8.1.2. The product form contains exactly the same number of nonzeros as \mathbf{L} and the ETMs are also lower triangular matrices with a nontrivial column vector formed from the corresponding columns of \mathbf{L} . An upper triangular matrix can also be written in product form. In this case the ETMs are upper triangular matrices but the nontrivial components are row vectors formed from the rows of \mathbf{U} . The newly created factors postmultiply the product. Therefore, in the product form they appear in the reverse order of their generation. If

$$\mathbf{L}^{-1} = \mathbf{L}^m \cdots \mathbf{L}^1$$

and

$$\mathbf{U}^{-1} = \mathbf{U}^1 \cdots \mathbf{U}^m$$

then

$$\mathbf{B}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1} = \mathbf{U}^1 \cdots \mathbf{U}^m \mathbf{L}^m \cdots \mathbf{L}^1, \quad (8.20)$$

where \mathbf{L}^i and \mathbf{U}^i , $i = 1, \dots, m$ are the ETMs of \mathbf{L}^{-1} and \mathbf{U}^{-1} , respectively. Note, the sequential number of generation of the entities is now given in superscripts for purposes that become clear in section 8.2.1.2. (8.20) shows that the inverse can be represented by a double product form. If there are logical variables in the basis they contribute to the product form with trivial unit matrices.

In practice there is no need to keep both \mathbf{L}^{-1} and \mathbf{U}^{-1} in product form because computing $\boldsymbol{\alpha}$ of (8.3) and \mathbf{h} of (8.4) can be organized differently as will be seen in section 8.2.3.

\mathbf{B} (or \mathbf{B}^{-1}) is in LU form just after a (re)factorization. When the basis changes \mathbf{B}^{-1} is updated. A straightforward method is to use the *product form update*, $\bar{\mathbf{B}}^{-1} = \mathbf{E}\mathbf{U}^1 \cdots \mathbf{U}^m \mathbf{L}^m \cdots \mathbf{L}^1$, where \mathbf{E} denotes the ETM created in the usual way from the column vector of the incoming variable. This method destroys the LU form and requires ‘hybrid’ procedures when solving equations in the simplex method. There are other updating methods that can restore triangularity after a basis change. These methods have the additional favorable feature that they help control the creation of new nonzeros in the factors. This possibility is not available with the pure product form update. The triangular update is more complicated and requires careful implementation. Section 8.2.1 presents a sparsity exploiting variant of the LU decomposition and in section 8.2.3 a method and its implementation are presented for maintaining triangularity during simplex iterations.

8.2.1 Determining a sparse LU form

While the LU form is due to Markowitz, the description of an efficient implementation of it is due to Suhl and Suhl [Suhl and Suhl, 1990]. Currently, it is widely accepted to be the best in its category. It is a healthy combination of the logical and numerical phase of the elimination and presents a good compromise between sparsity and numerical stability. In our presentation we follow the ideas of the quoted paper.

8.2.1.1 Determining triangular factors

The LU factorization always starts with identifying the triangular factors in \mathbf{B} . It can be done similarly as described in sections 8.1.2.3 and 8.1.3. The only difference is that the CC block in the PFI is permuted up to the top left corner. Figure 8.2 shows how the previous example of Figure 8.1 can be brought into this form. The general structure of the permuted matrix is shown in Figure 8.3. $\mathbf{U}^{(1)}$ and $\mathbf{L}^{(1)}$ denote the upper and lower triangular parts after the permutations. What is remaining is called the kernel and is denoted by \mathbf{K} .

In practice we can expect that there are very large triangular parts in sparse LP bases and the size of the kernel is very modest. Of course, there are instances where it is not true. The LU factorization is not particularly hindered by larger kernels while the PFI is very sensitive to it.

8.2.1.2 The algebra of the elimination

In the algebraic description of the elimination we will concentrate on sparsity and determine which positions change in an elimination step. In this way the unchanged positions need not be visited at all which leads to the minimization of the computational work. This does not come for free. There is an administrative overhead that has to be organized carefully.

The size of the kernel (obtained after the identification and permutation of the triangular parts) is denoted by ℓ . Obviously, $\ell \leq m$. Thus, $\mathbf{K} \in \mathbb{R}^{\ell \times \ell}$. The elimination is done in stages. In each stage the basic position of an unallocated column is determined and the row and column of the new pivot are permuted so that the pivot element goes into the next available position down the diagonal. At stage s the kernel is denoted by \mathbf{K}^s starting with $\mathbf{K}^1 = \mathbf{K}$. The number of elimination steps needed to complete the LU form is $\ell - 1$.

In the description the following notations will be used:

\mathcal{I}^s the set of row indices that have not been pivotal until stage s (available rows),

	\bar{C}		R		M	
\bar{C}	x x x	x			x x	9
	x			x		8
	x					1
R	x x					2
	x	x				3
	x x		x			4
M		x x x	x x x	x x x	x	5
	x			x x	x x x	6
		x x	x x	x x	x x	7
	9	8	1	2	3	4
						5
						6
						7

Figure 8.2. Matrix of Figure 8.1 rearranged in the form required by LU. \bar{C} denotes the reverse order of C . The original row and column indices are on the right and the bottom, respectively.

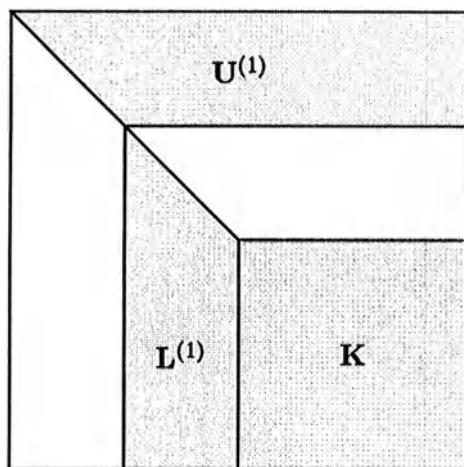


Figure 8.3. The general structure of the permuted sparse matrix before the LU procedure starts on the kernel. $U^{(1)}$ and $L^{(1)}$ denote the upper and lower triangular parts after the permutations. The shaded areas contain the nonzeros of the basis.

\mathcal{J}^s the set of column indices that have not been pivotal until stage s (available columns),

\mathbf{K}^s the active submatrix at stage s which has not been factorized yet,

ρ_j^i the elements of \mathbf{K}^s ,

$\bar{\rho}_j^i$ the elements of \mathbf{K}^{s+1} .

At stage s we choose a pivot element from among the nonzeros of \mathbf{K}^s . Assume it is $\rho_q^p \neq 0$, where the pivot position is $(p, q) \in \mathcal{I}^s \times \mathcal{J}^s$. In the resulting elimination step only those columns get transformed which have a nonzero in the pivot row. This is because the transformation is equivalent to multiplying the rest of \mathbf{K}^s by an ETM created from the pivot column. Therefore, the well known identity, namely, $\mathbf{E}\mathbf{a} = \mathbf{a}$ if \mathbf{a} has a zero in the pivot position of \mathbf{E} , applies to the elimination step. The columns to be transformed are identified from the rowwise list of the nonzeros in the pivot row p . Additionally, in the transformed columns only those elements change that are in the nonzero positions of the pivot column, as shown in Figure 8.4. The following sets help describe the transformation. Let

$$\mathcal{J}_p^s = \{j : \rho_j^p \neq 0, j \in \mathcal{J} \setminus \{q\}\}$$

denote the index set of those columns that have a nonzero in the pivot row and

$$\mathcal{I}_q^s = \{i : \rho_q^i \neq 0, i \in \mathcal{I} \setminus \{p\}\}$$

the index set of those rows that have a nonzero in the pivot column. Note, the position of the pivot is not included in the sets. From the above it follows that $\mathcal{I}^s \setminus \mathcal{I}_q^s$ is the index set of the zeros in the pivot column.

Elimination is achieved by adding appropriate multiples of a selected row (pivot row) to the other rows so that the coefficients in a selected column (pivot column) become zero, except in the pivot row. The active submatrix of stage $s + 1$ is determined according to the elimination formulas:

$$\bar{\rho}_j^i = \rho_j^i, \quad i \in \mathcal{I}^s \setminus \mathcal{I}_q^s, \quad j \in \mathcal{J}^s, \quad (8.21)$$

$$\bar{\rho}_j^i = \rho_j^i - \rho_j^p \frac{\rho_q^i}{\rho_q^p}, \quad i \in \mathcal{I}_q^s, \quad j \in \mathcal{J}_p^s. \quad (8.22)$$

The coefficients $-\rho_q^i/\rho_q^p$ are called multipliers that facilitate the elimination of the elements in the pivot column. If the pivot element is

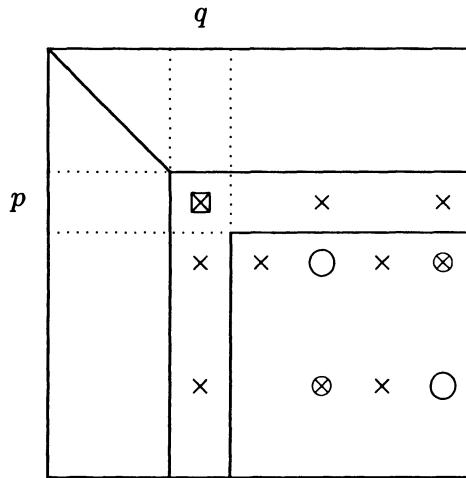


Figure 8.4. The elimination step highlighting fill-in. \boxtimes is the pivot element, \times denotes a nonzero entry, \otimes is a nonzero which is transformed and \circ is a fill-in. p and q denote the original row and column index of the pivot before permutation.

permuted to the next available diagonal position then the multipliers form a lower triangular ETM denoted by L^s which looks like

$$L^s = \begin{bmatrix} 1 & & & \\ \ddots & & & \\ & \times & & \\ & \vdots & \ddots & \\ & \times & & 1 \end{bmatrix},$$

where \times 's represent the multipliers except the diagonal element which is $1/\rho_q^p$. This L^s is the inverse of the elementary lower triangular matrix with column q of K^s as the non-trivial column.

During the current elimination step, fill-in is created in rows $i \in \mathcal{I}_q^s$ if $\rho_j^i = 0$ and $j \in \mathcal{J}_p^s$ (i.e., column j is subject to transformation). Clearly, if such a $\rho_j^i \neq 0$ then it is transformed according to (8.22). Under quite special circumstances a nonzero ρ_j^i can become zero after applying (8.22). To check whether it happened or not is usually more expensive than to automatically store a rarely emerging zero value.

After performing the elimination step we obtain

$$K^{s+1} = P^s L^s K^s Q^s = P^s L^s \cdots P^1 L^1 K Q^1 \cdots Q^s,$$

where the \mathbf{P} 's and \mathbf{Q} 's are permutation matrices responsible for the row and column permutations, respectively, needed to bring the pivot element to the next diagonal position. After $\ell - 1$ steps the kernel \mathbf{K}^ℓ is upper triangular.

8.2.1.3 Numerical considerations

To achieve sufficient numerical stability we have to ensure that magnitude of the chosen pivot elements is not too small relative to other nonzero elements in the active submatrix. This is *full pivoting* which generally guarantees stability. However, it would require checking all nonzeros and may end up with a choice that would result in substantial fill-in. As already mentioned in section 8.1.2.2, in sparse computing threshold pivoting has been viewed as a well performing compromise between numerical stability and sparsity. In the LU factorization we use the row version of it. As a reminder, it says that if a pivot row p has been selected in the active submatrix then an element ρ_q^p is a candidate to be a pivot if and only if it satisfies the

$$|\rho_q^p| \geq u \max\{ |\rho_j^p| : j \in \mathcal{J}_p^s \} \quad (8.23)$$

stability criterion, where u is the threshold parameter. (8.23) is nothing but the restatement of (8.9) using the notations of the current sparse case.

In each elimination step the growth of the entries in the active submatrix is bounded by the factor of $(1 + u^{-1})$, as a consequence of (8.22). This relationship makes it possible to determine an upper bound on the largest absolute element ρ_{\max} of the active submatrix during the elimination steps which is

$$\rho_{\max} \leq a \left(1 + \frac{1}{u}\right)^{\ell-1},$$

where a denotes the largest absolute element of \mathbf{K} , the matrix to be factorized numerically. ρ_{\max} can be used to estimate the accuracy of the LU factorization as recommended by Reid [Reid, 1971]. Let $\mathbf{LU} = \mathbf{PBQ} + \mathbf{F}$, where \mathbf{F} is a perturbation matrix. The magnitude of the elements of \mathbf{F} are bounded by

$$|f_j^i| \leq 3.01mg\rho_{\max},$$

where g is the relative accuracy of the number representation and m is the dimension of \mathbf{B} . In case of IEEE double precision numbers $g \approx 10^{-16}$. In sparse computing this bound is rather pessimistic, especially, if u is as low as 0.01 which is a realistic practical value.

8.2.1.4 Pivot strategy

In each step of the elimination the goal is to determine a pivot element that satisfies the (8.23) threshold pivoting criterion and also ensures a low fill-in (local strategy). To estimate the fill-in we use the m_j^i Markowitz merit numbers. It would be computationally expensive to check it for all nonzeros of \mathbf{K}^s . Therefore the search for a good pivot will be restricted. Many authors recommend to consider few rows with small row count. Suhl and Suhl [Suhl and Suhl, 1990] have pointed out that it is worth searching rows and columns alike with small counts. Therefore, no more than r rows/columns will be searched if the minimality of the Markowitz number can be guaranteed and more than r if a suitable pivot has not been found in the first r rows/columns. r is usually very small (2–4) and is kept as an adjustable parameter.

To describe the method of finding a pivot at stage s the following notation is introduced:

$$\begin{aligned}\mathcal{C}_k^s &= \{j : j \in \mathcal{J}^s \text{ and column } j \text{ of } \mathbf{K}^s \text{ has } k \text{ nonzeros}\} \\ \mathcal{R}_k^s &= \{i : i \in \mathcal{I}^s \text{ and row } i \text{ of } \mathbf{K}^s \text{ has } k \text{ nonzeros.}\}\end{aligned}$$

Furthermore, let M_j^s denote the minimum Markowitz number over all $\rho_j^i \neq 0$, $i \in \mathcal{I}^s$, that satisfy the (8.23) rowwise (!) threshold pivoting criterion. This is the smallest m_j^i over the eligible pivot elements in column j at stage s . It may happen that a column does not have an eligible pivot element. In this case we define $M_j^s = m^2$. Similarly, let M_i^s denote the minimum Markowitz number over all $\rho_j^i \neq 0$, $j \in \mathcal{J}^s$, which satisfy (8.23).

In Figure 8.5 we give a step by step description of the procedure that finds an acceptable pivot element with a low Markowitz count if one exists. It is to be noted that the procedure automatically identifies the hidden triangularity of the basis and places the corresponding columns/rows in the \mathbf{L} or \mathbf{U} section.

8.2.1.5 Controlling and monitoring numerical stability

The numerical stability of the procedure is controlled by the threshold pivot tolerance u and the *drop tolerance* ε_d . The latter one is used to set any computed value in \mathbf{L} or \mathbf{U} to zero if its magnitude is smaller than ε_d . Typical default values are $u = 0.01$ and $\varepsilon_d = 10^{-10}$ if IEEE (64 bit) double precision number representation is used. In case of the drop tolerance it is assumed that \mathbf{B} is scaled as part of scaling \mathbf{A} during preprocessing that finished with equilibration. The role of u is to strike a balance between sparsity and numerical stability. If u is small (< 0.01) then more elements can satisfy the threshold criterion, therefore, there

Procedure Find Pivot

```

If  $\mathcal{C}_1^s$  is nonempty then
    choose a column singleton from  $\mathcal{C}_1^s$  and return.
End If.

If  $\mathcal{R}_1^s$  is nonempty then
    choose a row singleton from  $\mathcal{R}_1^s$  as pivot and return.
End If.

 $\mu = m^2$ ,  $\nu = 0$ 
Do for  $k = 2$  to  $m$ 
    if  $\mathcal{C}_k^s$  is nonempty then
        do for all  $j \in \mathcal{C}_k^s$ 
            if  $M_j^s < \mu$  then
                 $\mu = M_j^s$ ,
                let  $i$  be the corresponding row index,
                record pivot position  $(i, j)$ ,
                if  $\mu \leq (k - 1)^2$  then return.
        End if.
         $\nu := \nu + 1$ ,
        if  $\nu \geq r$  and  $\mu < m^2$  then return.
    End do.
End if.

If  $\mathcal{R}_k^s$  is nonempty then
    do for all  $i \in \mathcal{R}_k^s$ 
        if  $M_i^s < \mu$  then
             $\mu = M_i^s$ ,
            let  $i$  be the corresponding column index,
            record pivot position  $(i, j)$ ,
            if  $\mu \leq k(k - 1)$  then return.
    End if.
     $\nu := \nu + 1$ ,
    if  $\nu \geq r$  and  $\mu < m^2$  then return.
End do.

End If.

End Do.

No acceptable pivot was found.

```

Figure 8.5. Pseudo-code of the Find Pivot procedure.

is more chance to find an element with low Markowitz number. On the opposite end, if $u = 1.0$ then the largest element is chosen. If it is unique then m_j^i is not considered at all, thus fill-in remains uncontrolled.

Suhl and Suhl [Suhl and Suhl, 1990] recommend to adjust the value of u dynamically if necessary. The way to do it is to monitor the numerical stability of the factorization. The computed largest absolute element in \mathbf{L} and \mathbf{U} is ρ_{\max} . If the ratio of ρ_{\max} to a (the largest absolute element in \mathbf{B}) exceeds a tolerance (useful values $10^{15} - 10^{16}$) the factorization is cancelled. If u was 0.01 then it is set to 0.1, otherwise it is set to $\min\{2u, 1.0\}$ and the factorization is restarted from the beginning or from the end of the triangularization. This latter requires quite a complicated administration and is usually not implemented.

8.2.2 Implementing the LU factorization

The efficient operation of the factorization heavily depends how it is implemented. Suhl and Suhl [Suhl and Suhl, 1990] have described important details that will be demonstrated in the following subsections.

8.2.2.1 Data structures

Data structures play a key role in the efficiency of the factorization. In most simplex implementations matrix \mathbf{A} is stored in columnwise packed form. The natural way \mathbf{B} is available for LU is also columnwise. However, the main operations of the elimination are better organized with rowwise representation of \mathbf{B} . Therefore, the basic columns of \mathbf{B} are extracted and stored as a set of packed row vectors in two parallel arrays, one for the indices and the other for the elements. The appropriate pointers to the rows are also set up. Some gap (space for, say up to 5, new elements) between the rows can be created to reduce the need for new copies of rows when fill-in occurs. In each row that has not been pivotal yet an element of largest magnitude is stored at the front. It removes the need for a search during threshold pivoting (8.23). For details about the packed representation of matrices, see Chapter 5.

The nonzero pattern of the columns is also stored as a set of packed vectors of the row indices. (The elements are not stored in this representation.) It provides an efficient access to each \mathcal{I}_q^s set. This is important for the easy identification of the rows that have to be transformed once the pivot has been determined.

At stage s of the elimination the array where \mathbf{B} was stored in the beginning of the procedure contains those rows of \mathbf{U} that were pivotal at stages $1, \dots, s-1$, and the active submatrix. The inverse of each lower triangular elementary transformation matrix \mathbf{L}^s is stored columnwise. They are accessed through a pointer to the beginning of each. The

elements stored here are the multipliers $-\rho_q^i/\rho_q^p$ and their row indices $i, i \in I_q^s$. The reciprocal of the pivot element is stored in front of the corresponding row vector in \mathbf{U} . By storing the multipliers, we actually store the inverse of a lower triangular elementary matrix. In general, it is easy to see, that if

$$\mathbf{L} = \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & a_p & \\ & & a_{p+1} & \\ & & \vdots & \ddots \\ & & a_m & 1 \end{bmatrix},$$

then

$$\mathbf{L}^{-1} = \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & \frac{1}{a_p} & \\ & & -\frac{a_{p+1}}{a_p} & \\ & & \vdots & \ddots \\ & & -\frac{a_m}{a_p} & 1 \end{bmatrix}$$

The only difference in our specific case is the storage of the diagonal element which is kept with the \mathbf{U} part.

The transformed \mathbf{B} and the nontrivial columns of the \mathbf{L}^{-1} matrices (the η vectors) are stored in the same array. The η -s of the \mathbf{L}^{-1} -s are written sequentially backward from the end of the arrays.

If a row cannot be updated in its current place plus the gap to the next row, it is moved to the next free area in the array. The space of the outdated copy of the row remains there as an unused memory area. If there is no contiguous free space to move a row a row compression is performed. It is often called *garbage collection*. It recovers the unused space of outdated and deleted rows. The nonzero pattern of columns may also expand. It is treated similarly, including the occasional column compression.

A recommended data structure for the LU factorization is shown in Figures 8.6 and 8.7. The sets of equal row and column counts, \mathcal{R}_k^s and \mathcal{C}_k^s are represented by doubly linked lists as described in sections 5.4 and

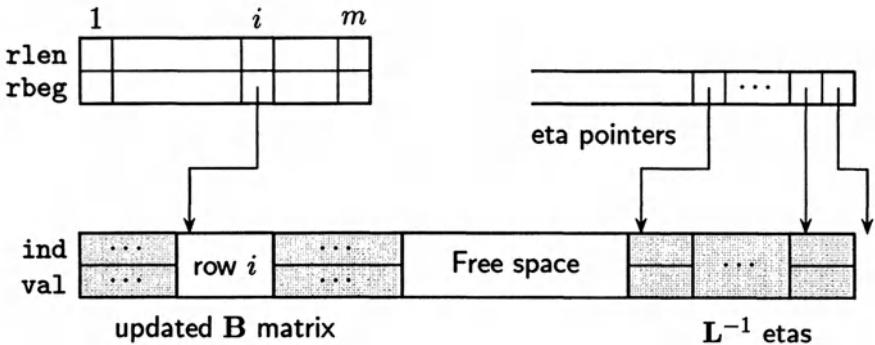


Figure 8.6. A rowwise storage scheme for the LU factorization. \mathbf{rlen} is row length (number of nonzeros), \mathbf{rbeg} is the beginning of the rows in the arrays \mathbf{ind} and \mathbf{val} , see Chapter 5. The etas are stored consecutively, thus their lengths are implied by the starting positions.

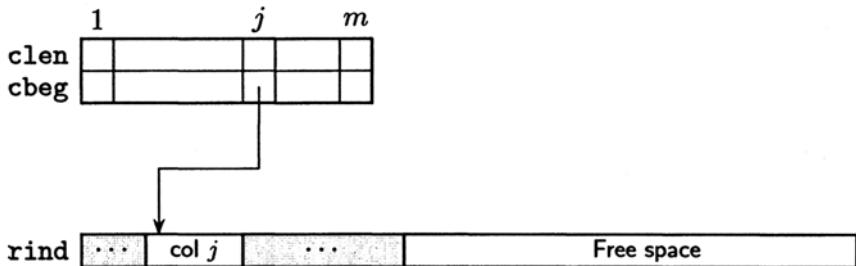


Figure 8.7. A scheme of the columnwise storage of row indices during LU factorization. $\mathbf{cлен}$ is column length (number of nonzeros), \mathbf{cbeg} is the beginning of the rows in the row index array \mathbf{rind} , see Chapter 5.

5.5. The size of the arrays containing the rowwise representation can be set to be equal to the nonzeros in the original matrix \mathbf{A} .

During factorization rows and columns are removed from the columnwise representation of the indices of nonzeros in \mathbf{B}^s . Therefore, it is shrinking in size and disappears at the end of the factorization.

8.2.2.2 Finding a pivot

According to the pivot finding algorithm of Figure 8.5 The search for a pivot starts with the first nonempty \mathcal{C}_k^s or \mathcal{R}_k^s , $k = 1, \dots, m$. If \mathcal{C}_1^s or \mathcal{R}_1^s is nonempty then a column or row singleton is found which uniquely

determines the pivot element. In any other case the threshold pivoting criterion (8.23) has to be checked. This requires access to the largest absolute element in the corresponding row of \mathbf{B}^s . To avoid search in such cases, this element is moved to the first position of the rows after the initial triangular factors have been identified. It can be done without a penalty as we have not assumed any specific order of the nonzeros of the rows. Later, if a row element is subject to a transformation the largest element is also checked and changed if necessary in that row. In this way, it is readily available for the threshold pivoting if we deal with a row from \mathcal{R}_k^s , $k > 1$. When the procedure works with \mathcal{C}_k^s , $k > 1$, then only the row indices of the nonzeros are known. Assume column j is examined and it has a nonzero in row i . The ρ_j^i element can be retrieved if we go through the list of nonzeros of row i and look for a matching column index j as in the rowwise representation we have both indices and values. On the average, this search requires the scanning of half of the nonzeros in a row. Having found ρ_j^i the threshold pivot criterion can easily be checked as the absolute largest element in each row is in the first position.

During the column search (examining \mathcal{C}_k^s) it can happen that none of the column nonzeros satisfies the threshold criterion in its corresponding row. In this case the column does not contain any eligible pivot element. Such a column is removed from \mathcal{C}_k^s for the rest of the factorization unless it becomes a column singleton in which case it is accepted. It is important that while such columns are not considered for pivoting they are updated during the elimination. The advantage of removing of ineligible columns is that their nonzeros do not take part in the above search. As search is, in general, an expensive operation, the saving in effort can be quite substantial if there are several temporarily rejected columns. These columns are processed after the ‘successful’ part of the elimination. The remaining submatrix is usually very small (if exists at all) and any stable pivoting procedure (like full pivoting) can be used to ensure stability. Sparsity is not considered at this stage.

8.2.2.3 Organizing the numerical elimination

Numerical elimination takes place if the selected pivot element is neither row nor column singleton. The new active submatrix is determined by (8.22).

A recommended way of organizing this part of the factorization is the following. When the pivot element is determined its reciprocal, $1/\rho_q^p$, is stored in the \mathbf{U} part at the front of row p . The main computational effort in (8.22) is adding a multiple of the sparse pivot row to the non-pivot sparse rows in \mathcal{I}_q^s . The participating rows are easily identified

by the columnwise storage of row indices. The elimination procedure uses a double precision work array \mathbf{w} of size m which is initialized to zero at the beginning of the elimination. After each step it is restored to zero so that the next step can reuse it, following the footsteps of Chapter 5. A boolean (or integer) array \mathbf{t} is also needed, initialized to zero, for marking some positions. Of particular interest is the fill-in. The number of positions in a non-pivotal row that may be subject to fill-in is denoted by f . Experience has shown that there is a quite high chance of having no more than one fill-in. It can be accommodated in the current representation of a row as its element in the pivot column need not be stored for the next stages. Therefore, the case of $f = 1$ is given a special treatment. Below we give a step by step description of the elimination procedure.

Procedure LU Elimination

Step 0. Initialization. Assume, pivot position (p, q) has been determined. Expand the pivot row without the pivot element itself into \mathbf{w} , i.e., copy the ρ_j^p , $j \in \mathcal{J}_r^s$ elements to the positions of their column indices. Set the marker t_j of the nonzero positions to 1.

Step 1. Do for all i , $i \in \mathcal{I}_q^s$

- 1 Set f to the number of indices in \mathcal{J}_p^s . Compute the multiplier of row i : $-\rho_q^i / \rho_q^p$ and store it in the next position in the \mathbf{L} section. Copy the element in the last position of row i to the position of ρ_j^p (that was the element in the pivot column). Decrement the row count by 1: $r_i := r_i - 1$.
- 2 Scan the sparse row i .

For each $\rho_j^i \neq 0$ in row i compute $\bar{\rho}_j^i$ according to (8.22). If element ρ_j^p in the pivot row is nonzero (check $w_j \neq 0$) decrement f by 1: $f := f - 1$, unmark column j by setting $t_j = 0$. If $w_j = 0$ then ρ_j^i remains unchanged.

Now f is an upper bound on the number of fill-in in row i .

- 3 If $f > 0$ then

If $f > 1$ then

Copy the updated row i to a free space area.

If there is not enough contiguous free space to store the updated row i plus the fill-in perform row compression.

End if.

Scan the pivot row to compute the fill-in elements.

```

Do for all  $j$ ,  $j \in \mathcal{J}_p^S$ 
  If  $t_j = 0$  then
     $t_j = 1$ 
  Else
    Compute  $\bar{\rho}_j^i$  according to (8.22).
    Store  $\bar{\rho}_j^i$  at the end of the current copy of row  $i$ ,
    or in the original place if  $f = 1$  (just one fill-in).
    Store row index  $i$  in the column representation.
    Compress column if necessary. Update column counts.
  End if.
End do.
Else
  Scan the pivot row and set  $t_j = 1$  for  $j \in \mathcal{J}_p^s$ .
End if.
End do.

```

Step 2. Restore the work array w and the boolean array t to zero by scanning the nonzero pattern of the pivot row as only these positions were involved in the operations, i.e., $j \in \mathcal{J}_r^s$.

Some details are omitted in the above description, like placing the largest absolute element in the front of the row, checking if there is enough space for the fill-in in the array, or deleting row and column indices with changed counts from the sublists and placing them in the proper lists, i.e., updating the \mathcal{I}_k^s and \mathcal{J}_k^s index sets of equal row and column counts. This latter operations are easily performed by the technique described in Chapter 5.

If $f = 0$ in the procedure then scanning the nonzeros of the pivot row is skipped. If $f = 1$ the fill-in element can be stored at the end of the non-pivot row without any further checking. There are experimental observations that $f > 1$ occurs in less than 20% of cases for large scale LP problems.

Some points of the procedure can be highlighted by a small example where only the pivot row p and a general row i are represented. It is to be noted that the pivot row p is in expanded form during the procedure while row i is packed, though, for simplicity, it is also shown in full.

	q	j_1	j_2	j_3		
p	x		x			x
i	x			x		x

In Step 1(b) positions j_2 and j_3 of row i are transformed according to (8.22). In Step 1(c) (scanning the pivot row) position j_1 is transformed which results in a fill-in. The markers help skipping the already updated elements in row i . At the end of processing row i the markers are restored to identify all nonzeros of the pivot row. In Step 2 both auxiliary arrays (\mathbf{w} and \mathbf{t}) are restored to zero. It is done by visiting only the nonzero positions of the pivot row: $j \in \mathcal{J}_r^s$.

8.2.3 Maintaining triangularity during iterations

With the LU decomposition of the basis matrix the main operations of type (8.3) and (8.4) can be organized in the following way. After the LU procedure presented in section 8.2.1, ignoring permutations, we have the form

$$\mathbf{L}^{-1}\mathbf{B} = \mathbf{U}. \quad (8.24)$$

In practical terms it means we have computed (and stored) \mathbf{U} and \mathbf{L}^{-1} (\mathbf{L}^{-1} in product form). The solution of $\mathbf{B}\boldsymbol{\alpha} = \mathbf{a}$ for $\boldsymbol{\alpha}$ can formally be obtained as $\boldsymbol{\alpha} = \mathbf{B}^{-1}\mathbf{a} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{a}$. With notation $\bar{\boldsymbol{\alpha}} = \mathbf{L}^{-1}\mathbf{a}$ we can write $\mathbf{U}\boldsymbol{\alpha} = \bar{\boldsymbol{\alpha}}$. For traditional reasons we maintain the FTRAN acronym for computing $\boldsymbol{\alpha}$ in this way. It can now be performed in two steps.

- FTRANL: Compute $\bar{\boldsymbol{\alpha}} = \mathbf{L}^{-1}\mathbf{a}$,
- FTRANU: Solve $\mathbf{U}\boldsymbol{\alpha} = \bar{\boldsymbol{\alpha}}$ for $\boldsymbol{\alpha}$.

For similar traditional reasons, computing $\boldsymbol{\pi}$ from $\mathbf{B}^T\boldsymbol{\pi} = \mathbf{h}$ is called BTRAN. It can be conceived as $\mathbf{h} = \mathbf{B}^T\boldsymbol{\pi} = \mathbf{U}^T\mathbf{L}^T\boldsymbol{\pi} = \mathbf{U}^T\bar{\boldsymbol{\pi}}$ if notation $\mathbf{L}^T\boldsymbol{\pi} = \bar{\boldsymbol{\pi}}$ is used. From the latter $\boldsymbol{\pi} = (\mathbf{L}^{-1})^T\bar{\boldsymbol{\pi}}$. Therefore, BTRAN can be performed in the following two steps.

- BTRANU: Solve $\mathbf{U}^T\bar{\boldsymbol{\pi}} = \mathbf{h}$ for $\bar{\boldsymbol{\pi}}$.
- BTRANL: Compute $\boldsymbol{\pi} = (\mathbf{L}^{-1})^T\bar{\boldsymbol{\pi}}$.

As a ‘result’ of the first basis change after an LU factorization has been performed, triangularity may disappear. However, there are methods that can restore triangularity of the decomposition during the simplex iterations. The list of contributors of important developments in this area include Bartels and Golub [Bartels and Golub, 1969], Forrest and Tomlin [Forrest and Tomlin, 1972], Saunders [Saunders, 1976], Goldfarb [Goldfarb, 1977], Reid [Reid, 1982], Suhl and Suhl [Suhl and Suhl, 1993]. They emphasize different aspects of triangular LU update, like numerical stability, computational efficiency in case of limited memory and, lately, overall efficiency. Suhl and Suhl have proposed a remarkable

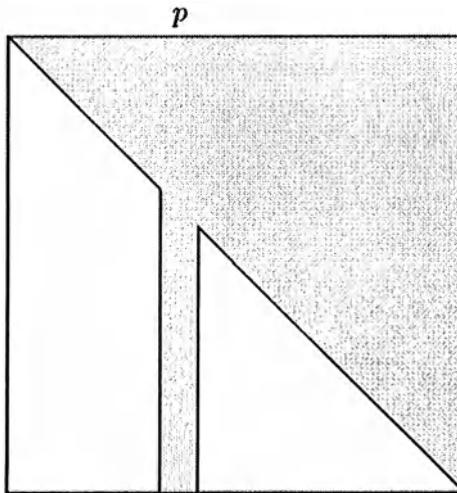


Figure 8.8. The spike column $\mathbf{g} = \mathbf{L}^{-1} \mathbf{a}_q$ in position p destroys the triangularity of \mathbf{U} .

algorithm that represents just a small compromise in numerical stability while achieving high computational efficiency, memory allowing. In this section we present the method they have developed. It is important to emphasize that all other methods are also suitable for inclusion in a simplex implementation using LU factorization.

8.2.3.1 The logic of LU update

Assume the column vector of the incoming variable \mathbf{a}_q replaces the p -th basic variable (with the corresponding column vector $\mathbf{B}\mathbf{e}_p$). The new basis can formally be written as

$$\bar{\mathbf{B}} = \mathbf{B} + (\mathbf{a}_q - \mathbf{B}\mathbf{e}_p)\mathbf{e}_p^T. \quad (8.25)$$

If (8.25) is premultiplied by \mathbf{L}^{-1} we obtain

$$\mathbf{L}^{-1}\bar{\mathbf{B}} = \mathbf{U} + (\mathbf{L}^{-1}\mathbf{a}_q - \mathbf{U}\mathbf{e}_p)\mathbf{e}_p^T. \quad (8.26)$$

In this form the permutations are omitted for easier understanding of the essence of the procedure. (8.26) says that in order to get the new $\mathbf{L}^{-1}\bar{\mathbf{B}}$, column p of \mathbf{U} must be replaced by a vector \mathbf{g} which is derived from the incoming vector \mathbf{a}_q as $\mathbf{g} = \mathbf{L}^{-1}\mathbf{a}_q$. Vector \mathbf{g} is referred to as *spike*. The name becomes obvious by looking at the matrix in Figure 8.8 which represents $\mathbf{L}^{-1}\bar{\mathbf{B}}$. It is also said that this matrix is *spiked*.

In general, the spike is also a sparse vector, in particular if \mathbf{L}^{-1} itself is sparse. Therefore, the last nonzero entry of the spike may not be at

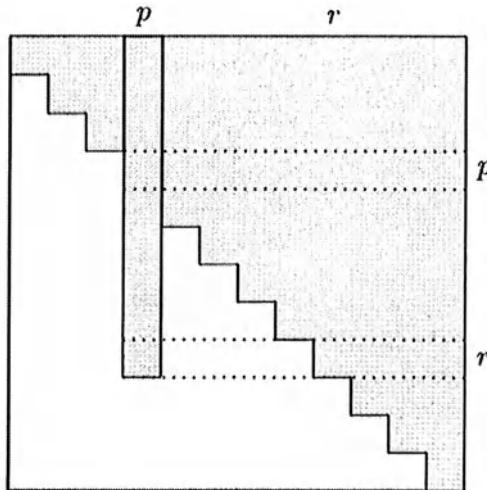


Figure 8.9. A sparse spike in column p has its last nonzero in row r .

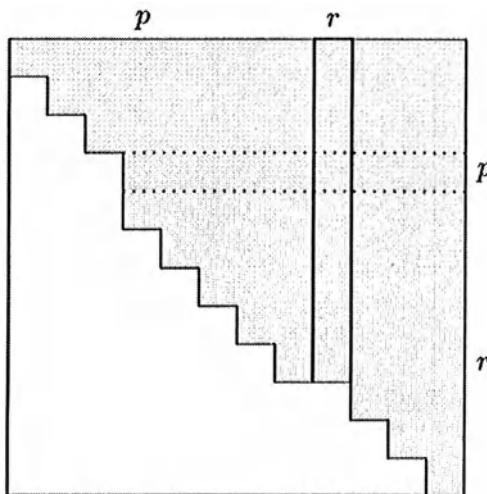


Figure 8.10. The upper Hessenberg matrix obtained by moving the spike to position r while shifting the other columns to the left by one to fill the gap.

the bottom of the matrix as in Figure 8.8 but rather in a position $r < m$ as shown in Figure 8.9.

The key idea in sparse LU update is the utilization of the sparsity of the spike column. Suhl and Suhl permute the spike to column r which corresponds to the position of its last nonzero element. As a result we obtain an upper Hessenberg matrix illustrated in Figure 8.10.

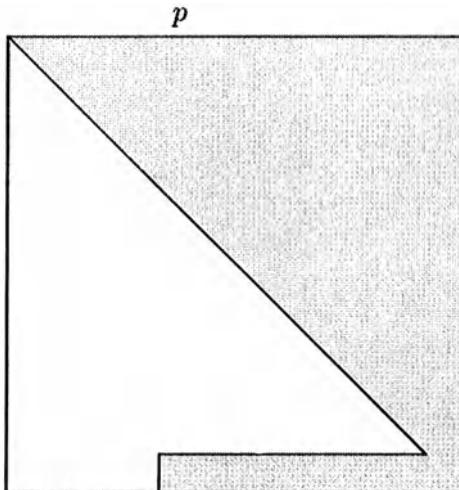


Figure 8.11. Row-spiked structure of the Forrest-Tomlin update scheme.

This is in contrast with the Forrest-Tomlin scheme [Forrest and Tomlin, 1972] which permutes the spike to the last column and then permutes row p to the last row leading to a spiked row structure shown in Figure 8.11 which is not a Hessenberg matrix. The Forrest-Tomlin method eliminates the spike row by adding suitable multiples of rows to the spike. In this way there is no fill-in in the \mathbf{U} part and all eliminations can be stored in the \mathbf{L} part.

Returning to the Suhl-Suhl method, during the LU factorization the columns of the basis have been permuted such that it is possible to pivot down the diagonal. The actual pivot order is contained in a permutation matrix \mathbf{R} . During the triangular LU update symmetric row and column permutations are performed so that the diagonal elements remain the pivots. As a result, it is sufficient to maintain just one permutation matrix \mathbf{R} .

If the original row indices are kept in the \mathbf{L} part and the permutations are applied to the \mathbf{U} part then the spike $\mathbf{L}^{-1}\mathbf{a}_q$ can be computed and inserted without permutations. If there is a basis change, a further permutation matrix is applied to the spiked \mathbf{U} , followed by some eliminations, to restore its triangularity. At iteration k the representation is

$$(\mathbf{L}^k)^{-1}\mathbf{B}^k = \mathbf{R}^k\mathbf{U}^k(\mathbf{R}^k)^T,$$

where \mathbf{B}^k is the basis matrix, \mathbf{R}^k is the permutation matrix, and \mathbf{L}^k and \mathbf{U}^k are lower and upper triangular matrices, respectively.

Now, we assume that there is sufficient computer memory available such that the indices and values of nonzeros of \mathbf{U} can be stored row- and columnwise, while \mathbf{L}^{-1} is stored columnwise. It would be sufficient to store the values just once but it would result in a non-sequential access to elements that can seriously deteriorate the cache performance of the implementation. The usefulness of the double storage of \mathbf{U} will become clear when the details of the operations with the LU form are discussed in section 8.2.4. The LU update procedure can be described by the following steps.

Procedure LU Update

Step 0. Initialize. Assume spike $\mathbf{L}^{-1}\mathbf{a}_q$ has been computed and pivot row p determined.

Step 1. Remove column p from \mathbf{U} .

Step 2. Perform column permutation. First, determine position r , the last nonzero of the spike, see Figure 8.9. Move columns $p, p+1, \dots, r$ one position to the left leaving position r free. Place the spike column in position r . As a result, we obtain the structure shown in Figure 8.10.

Step 3. Use rows $p+1, \dots, r$ to eliminate entries in positions $p, p+1, \dots, r-1$ of row p . The result is shown in Figure 8.12. Store the multipliers in the L part as a triangular elementary row transformation matrix.

Step 4. Perform row permutation. Place row p in position r after moving rows $p+1, \dots, r$ one position up. In this way triangularity of \mathbf{U} is restored, see Figure 8.13.

Step 5. Store the modified row (initially row p , now row r) back to the sparse row- and columnwise representation of \mathbf{U} .

In general, at iteration k after a refactorization we have an upper triangular \mathbf{U}^k and a lower triangular $(\mathbf{L}^k)^{-1}$. The former is stored as a collection of sparse row and column vectors, the latter is stored as sparse column (from the factorization) and row (from the LU update) etas.

8.2.3.2 Implementing the LU update

In this section we comment on the actual steps of the LU update procedure to point out some important details of its implementation. The most difficult part is the update of the row- and columnwise representation of \mathbf{U} . Efficiency of the update is achieved by the use of sophisticated

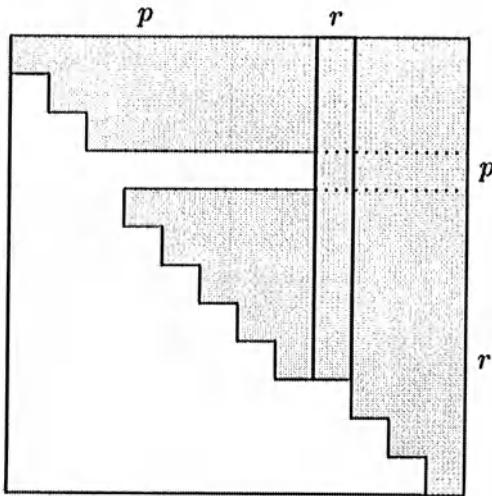


Figure 8.12. The matrix after positions $p, p + 1, \dots, r - 1$ of row p have been eliminated.

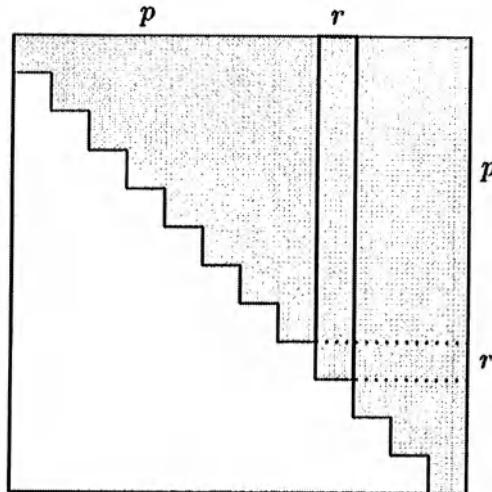


Figure 8.13. Triangularity of \mathbf{U} is restored after rows $p + 1, \dots, r$ have been moved one position up.

data structures. They require extra storage space that is assumed to be available. The specific requirements will be assessed.

During the computation of the spike $\mathbf{g} = \mathbf{L}^{-1}\mathbf{a}_q$ the indices of the nonzero positions of \mathbf{g} are stored in a list for later use during the elimination in Step 3. Obviously, the length of the list cannot exceed m .

Removing column p of \mathbf{U} in Step 1 is a complicated operation because it involves the removal of the nonzeros of \mathbf{U} from both the row- and columnwise representation. It can be done efficiently at the expense of some more administration (which requires extra space and logic). Two pointer arrays are introduced. For each element in one representation (row- or columnwise) of \mathbf{U} there is a pointer to the position where this element is stored in the other representation. For deletion of an element from a packed row or column form the method described on page 75 can be used. When an element is deleted from one representation its counterpart on the other one is available through the extra pointer and can also be deleted.

The column permutation in Step 2 is performed logically (recording the movements in a permutation matrix which is stored in a vector form) without physically moving the elements of the columns.

Step 3 of the LU update procedure can be performed efficiently by Method-2 described on page 73 by scattering row p of \mathbf{U} in a work array \mathbf{w} . The list of the indices of the resulting nonzero entries is preserved.

The row permutation in Step 4 is the same as the column permutation in Step 2 (symmetric row and column permutations).

In Step 5 the packed form of the new row r of \mathbf{U} is obtained by a gather operation from \mathbf{w} . It can simply be inserted in (added to) the rowwise form of \mathbf{U} . The columnwise representation can also directly (without search) be updated by scanning the column indices of the nonzero entries and inserting them in the packed form of the appropriate columns using techniques discussed in section 5.3.

In summary, after the LU update we have a row- and columnwise stored \mathbf{U} and a new elementary row matrix that premultiplies the product form of \mathbf{L}^{-1} .

8.2.4 Operations with the LU form

The main operations with the LU form are FTRAN and BTRAN can substantially benefit from the double representation of \mathbf{U} . More specifically, it becomes possible to take advantage of the sparsity of the vectors to be FTRAN-ed and BTRAN-ed and save, usually a substantial amount of, operations.

For easier reference, we repeat the mathematical form of FTRAN and BTRAN using a general vector \mathbf{h} and equation (8.24). The permutation matrix is also ignored for notational simplicity.

FTRAN: Solve $\mathbf{B}\gamma = \mathbf{h}$ for γ . It is done in two steps.

$$\text{FTRANL: Compute } \bar{\mathbf{h}} = \mathbf{L}^{-1}\mathbf{h}, \quad (8.27)$$

$$\text{FTRANU: Solve } \mathbf{U}\gamma = \bar{\mathbf{h}} \text{ for } \alpha. \quad (8.28)$$

BTRAN: Solve $\mathbf{B}^T\gamma = \mathbf{h}$ for γ . It is also done in two steps.

$$\text{BTRANL: Solve } \mathbf{U}\bar{\mathbf{h}} = \mathbf{h} \text{ for } \bar{\mathbf{h}}, \quad (8.29)$$

$$\text{BTRANU: Compute } \gamma = (\mathbf{L}^{-1})^T\bar{\mathbf{h}}. \quad (8.30)$$

One step of FTRANL is of type $\bar{\mathbf{a}} = \Lambda^{-1}\mathbf{a}$, where Λ^{-1} is an elementary lower triangular matrix ETM with η denoting its nontrivial column. If it is a columnwise ETM, we have

$$\begin{bmatrix} \bar{a}_1 \\ \vdots \\ \bar{a}_p \\ \vdots \\ \bar{a}_m \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & \eta^p & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_p \\ \vdots \\ a_m \end{bmatrix}, \quad (8.31)$$

where

$$\bar{a}_i = \begin{cases} a_i & \text{for } i = 1, \dots, p-1, \\ a_p \eta^p & \text{for } i = p, \\ a_i + a_p \eta^i & \text{for } i = p+1, \dots, m, \end{cases} \quad (8.32)$$

which entails that $\bar{\mathbf{a}} = \mathbf{a}$ if $a_p = 0$. It means that the entire operation with this ETM can be skipped saving the retrieval of elements and operations on them (with known result). Obviously, this observation is a special case of (3.2).

If the elementary matrix is a rowwise ETM then

$$\begin{bmatrix} \bar{a}_1 \\ \vdots \\ \bar{a}_p \\ \vdots \\ \bar{a}_m \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & \eta_p & \dots & \eta_m \\ & & & \ddots & \\ & & & & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_p \\ \vdots \\ a_m \end{bmatrix}, \quad (8.33)$$

where

$$\bar{a}_i = \begin{cases} a_i & \text{for } i \neq p, \\ \sum_{i=p}^m a_i \eta_i & \text{for } i = p. \end{cases} \quad (8.34)$$

This operation involves all elements of the eta vector of the ETM. No saving can be achieved.

In FTRANL the elementary matrices are applied in the order of their generation. Therefore, as long as we are using the column etas generated during the LU factorization we can take advantage of the saving offered by (8.31) and (8.32) which can be quite substantial in case of sparse vectors. When row etas are used then such an advantage is not available, see (8.33) and (8.34).

Analyzing one step of BTRANL which is computed as $\bar{\mathbf{a}}^T = \mathbf{a}^T \Lambda^{-1}$ we can observe the following. If Λ^{-1} is a columnwise ETM then

$$\bar{a}_i = \begin{cases} a_i & \text{for } i \neq p, \\ \sum_{i=p}^m a_i \eta^i & \text{for } i = p, \end{cases}$$

which is the same as (8.34), so is the conclusion. However, if Λ^{-1} is a rowwise ETM then

$$\bar{a}_i = \begin{cases} a_i & \text{for } i = 1, \dots, p-1, \\ a_p \eta_p & \text{for } i = p, \\ a_i + a_p \eta_i & \text{for } i = p+1, \dots, m, \end{cases}$$

which is identical with (8.32), i.e., a saving is possible if $a_p = 0$.

Operations with \mathbf{U} can also be analyzed. Consider first FTRANU, $\mathbf{U}\boldsymbol{\alpha} = \mathbf{a}$.

$$\begin{bmatrix} u_1^1 & \dots & u_i^1 & \dots & u_m^1 \\ \ddots & \vdots & & & \vdots \\ & u_i^i & \dots & u_m^i & \\ & \ddots & \vdots & & \\ & u_m^m & & & \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_i \\ \vdots \\ \alpha_m \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_i \\ \vdots \\ a_m \end{bmatrix},$$

from which $\boldsymbol{\alpha}$ can be determined by back-substitution

$$\alpha_i = \frac{1}{u_i^i} \left(a_i - \sum_{j=i+1}^m u_j^i \alpha_j \right), \quad i = m, m-1, \dots, 1.$$

This equation shows that when an α_j value has become known it multiplies the elements of the same column in \mathbf{U} . Thus, if $\alpha_j = 0$ the entire column j can be skipped when the contributions of the columns are accumulated going backwards. Therefore, this operation can benefit from the columnwise access to \mathbf{U} .

In BTRANU we use the transpose of \mathbf{U} to solve $\mathbf{U}^T \boldsymbol{\alpha} = \mathbf{a}$. In this case we have

$$\begin{bmatrix} u_1^1 & & & \\ \vdots & \ddots & & \\ u_i^1 & \dots & u_i^i & \\ \vdots & & \vdots & \ddots \\ u_m^1 & \dots & u_m^i & \dots & u_m^m \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_i \\ \vdots \\ \alpha_m \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_i \\ \vdots \\ a_m \end{bmatrix},$$

and $\boldsymbol{\alpha}$ can be determined by forward substitution

$$\alpha_i = \frac{1}{u_i^i} \left(a_i - \sum_{j=1}^{i-1} u_j^i \alpha_j \right), \quad i = 1, 2, \dots, m.$$

Here we can observe that once the value of an α_j has been determined, it multiplies the elements of row j only. It entails that if an $\alpha_j = 0$ row j can be skipped when the values of the unknowns are accumulated downwards by rows. Therefore, this operation can benefit from the rowwise access to \mathbf{U} .

As we have both row- and columnwise representation of \mathbf{U} we can always use the one which can result in saving in the computations. In practice this saving is usually substantial. This is why the above techniques are worth implementing.

It is important to remember that the LU factorization generates only a rowwise representation of \mathbf{U} . Therefore, if the Suhl-Suhl LU update is used the columnwise structure of \mathbf{U} must be constructed after the factorization.

8.3. Concluding remarks

It is absolutely essential to have an LU or a PFI representation of the basis or its inverse which is numerically stable, has very few nonzeros and can be obtained quickly. Therefore, it is obvious why (re)inversion and (re)factorization attracted much attention in the last decades of the XXth century. Several researchers have contributed to the development of new and more capable methods. The development was triggered by the increasing need for solving large scale LP problems safely and efficiently. It was also inspired by the new possibilities offered by the evolution of computer technology.

In addition to the references already given in this chapter, interested readers can find more information about the product form in [Hadley, 1962] and [Orchard-Hays, 1968]. The development of the LU factoriza-

tion can be followed by studying [Hellerman and Rarick, 1971], [Hellerman and Rarick, 1972] and [Erismann et al., 1985].

Further interesting aspects of the LU and updating can be found in [Brown and Olson, 1993] and [Bisschop and Meeraus, 1977].

Chapter 9

THE PRIMAL ALGORITHM

The first version of the primal simplex method (PSM-1) was described in Chapter 2. It was able to handle problems with type-2 variables only. This chapter extends PSM-1 in several ways. The new algorithm will be referred to as PSM-G, where ‘G’ stands for ‘general’.

First of all, PSM-G will handle all types of variables in computational form #1 (CF-1). Furthermore, every algorithmic part of it will be discussed in such detail that is sufficient for successful implementation.

As computational form #2 (CF-2) differs just slightly from CF-1, the presented algorithm will require only small modifications to handle problems given in CF-2. The necessary changes will be explained. Therefore, the ideas presented in this chapter make it possible to prepare an advanced implementation of either formulation.

In addition to the new types of variables a substantial difference between PSM-1 and the forthcoming algorithm is that in case of the latter sparsity will be given full consideration.

This chapter presents a comprehensive treatment of the algorithmic techniques of the simplex method. It includes details needed for successful implementation. As a result, readers with suitable programming experience can develop a state-of-the-art version of the primal method. The dual method is discussed in a similar depth in Chapter 10.

Recalling CF-1, the LP problem is the following

$$\min \quad z = \mathbf{c}^T \mathbf{x} \tag{9.1}$$

$$\text{s.t.} \quad \mathbf{A}\mathbf{x} = \mathbf{b}, \tag{9.2}$$

$$\boldsymbol{\ell} \leq \mathbf{x} \leq \mathbf{u}, \tag{9.3}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and the vectors are of compatible dimensions. The finite lower bounds in ℓ are equal to zero. The types of the components of \mathbf{x} are defined as

ℓ_j		u_j	Type	
0	\leq	x_j	\leq	0
0	\leq	x_j	\leq	u_j
0	\leq	x_j	\leq	$+\infty$
$-\infty$	\leq	x_j	\leq	$+\infty$
				3

(9.4)

9.1. Solution, feasibility, infeasibility

For CF-1, the notions of Chapter 2 need to be revised to extend their definitions for all types of variables appearing in (9.4).

An \mathbf{x} is called a *solution* if it satisfies the (9.2) joint constraints. If a solution satisfies (9.3) (which is the set of type constraints in disguise) it is called a *feasible solution*. The set of feasible solutions is denoted by \mathcal{X} , i.e.,

$$\mathcal{X} = \{\mathbf{x} : \mathbf{Ax} = \mathbf{b}, \ell \leq \mathbf{x} \leq \mathbf{u}\}.$$

Feasible solutions form a convex set (for proof see section 2.1). If \mathcal{X} is empty we say the *problem is infeasible*.

A feasible solution \mathbf{x}^* is called *optimal* if $\mathbf{c}^T \mathbf{x}^* \leq \mathbf{c}^T \mathbf{x}$ for all $\mathbf{x} \in \mathcal{X}$. There may be many optimal solutions with the same objective value. If the optimum is not unique we say the problem has *alternative optima*. Any convex linear combination of optimal solutions is also an optimal solution. Proof is similar to that of in (2.5).

The index set of variables is $\mathcal{N} = \{1, \dots, n\}$, as in section 2.1. Since \mathbf{A} is of full row rank m linearly independent columns $\mathbf{a}_{k_1}, \dots, \mathbf{a}_{k_m}$ can be selected to form a basis of \mathbb{R}^m . The index set of these columns is denoted by \mathcal{B} ,

$$\mathcal{B} = \{k_1, \dots, k_m\}.$$

\mathcal{B} is also referred to as *basis heading*. It defines the relationship between the basic positions and the indices of the variables in the basis. The index set of the ‘remaining’ (nonbasic) variables is denoted by \mathcal{R} . Sometimes \mathcal{B} and \mathcal{R} will be used as subscripts to vectors and matrices with the same meaning as in section 2.1. The notions of *basic* and *nonbasic variables* are also the same.

Similarities go even further. Recalling the basic/nonbasic partitioning of the matrix and vectors in (2.7) and (2.8), $\mathbf{Ax} = \mathbf{b}$ can be written as

$\mathbf{Bx}_\mathcal{B} + \mathbf{Rx}_\mathcal{R} = \mathbf{b}$, from which

$$\mathbf{x}_\mathcal{B} = \mathbf{B}^{-1} (\mathbf{b} - \mathbf{Rx}_\mathcal{R}). \quad (9.5)$$

It is still true that the nonbasic variables uniquely determine the values of the basic variables. Now, however, nonbasic variables are not necessarily equal to zero, namely, some type-1 nonbasic variables can be at their upper bound. Formally, let \mathcal{U} denote the index set of such variables, i.e.,

$$\mathcal{U} = \{k : k \in \mathcal{R}, x_k = u_k\}. \quad (9.6)$$

Thus, taking (9.5) and (9.6) into account, a *basic solution* is determined by two index sets, namely, \mathcal{B} and \mathcal{U} in the following way

$$\mathbf{x}_\mathcal{B} = \mathbf{B}^{-1} \left(\mathbf{b} - \sum_{k \in \mathcal{U}} u_k \mathbf{a}_k \right). \quad (9.7)$$

Introducing notation

$$\mathbf{b}_\mathcal{U} = \mathbf{b} - \sum_{k \in \mathcal{U}} u_k \mathbf{a}_k,$$

(9.7) can be written as

$$\mathbf{x}_\mathcal{B} = \mathbf{B}^{-1} \mathbf{b}_\mathcal{U}, \quad (9.8)$$

which resembles to (2.10) very much. It is to be noted that in (9.7) all nonbasic variables (whether in \mathcal{U} or not) are at feasible level. If $\mathbf{\ell}_\mathcal{B} \leq \mathbf{x}_\mathcal{B} \leq \mathbf{u}_\mathcal{B}$ then $\mathbf{x}_\mathcal{B}$ is called a *basic feasible solution* (BFS). We also say that the solution defined by \mathcal{B} and \mathcal{U} is feasible.

Let $\mathcal{I} = \{1, \dots, m\}$ denote the set of all positions in the basis. It can be decomposed by the types of basic variables into

$$\mathcal{I} = \mathcal{I}_0 \cup \mathcal{I}_1 \cup \mathcal{I}_2 \cup \mathcal{I}_3,$$

where \mathcal{I}_j , $j = 0, 1, 2, 3$, contains those basic positions i for which the type of the corresponding basic variable is j , i.e., $\text{type}(x_{k_i}) = j$.

If there is at least one basic variable at infeasible level the basic solution is called *infeasible*. Obviously, a type-3 variable is always at a feasible level. Other variables can violate their individual bounds either by being negative, or with a value greater than their finite upper bound (where applicable). Taking this into account, two sets of infeasible basic positions can be defined,

$$\mathcal{M} = \{i : x_{k_i} < 0, i \in \mathcal{I}_0 \cup \mathcal{I}_1 \cup \mathcal{I}_2\}, \quad (9.9)$$

and

$$\mathcal{P} = \{i : x_{k_i} > u_{k_i}, i \in \mathcal{I}_0 \cup \mathcal{I}_1\}. \quad (9.10)$$

A type-0 variable with $x_{k_i} = 0$ is simultaneously at lower and upper bound. The index set of positions of feasible basic variable is

$$\mathcal{F} = \mathcal{I} \setminus (\mathcal{M} \cup \mathcal{P}). \quad (9.11)$$

If $\mathcal{M} \cup \mathcal{P} = \emptyset$ the basis (and also the BFS) is said to be feasible.

The traditional definition of *degeneracy* also need to be extended. A basic solution is said to be degenerate if at least one of the basic variables is at one of its bounds. Clearly, a type-3 basic variable can never cause degeneracy even if it is equal to zero. On the other hand, if a type-1 basic variable is at its upper bound the basis is degenerate. More formally, let \mathcal{D} denote the set of degenerate positions

$$\mathcal{D} = \{i : (x_{k_i} = 0, i \in \mathcal{I}_0 \cup \mathcal{I}_1 \cup \mathcal{I}_2) \vee (x_{k_i} = u_{k_i}, i \in \mathcal{I}_1)\}. \quad (9.12)$$

A basic solution is *nondegenerate* if $\mathcal{D} = \emptyset$, otherwise it is *degenerate*. The *degree of degeneracy* is the number of basic variables at bound, which is $|\mathcal{D}|$.

Computationally, the main problem with degeneracy is that if a basic variable is at its bound it can make the steplength of the incoming variable zero. This situation will be further investigated during the discussion of the general ratio test.

EXAMPLE 9.1 Assume $\mathcal{B} = \{7, 4, 9, 2, 1, 3\}$ is a set of basic variables which are subject to the following individual bounds (given in this order):

$$\begin{array}{rclcl} 0 & \leq & x_7 & \leq & +\infty \\ 0 & \leq & x_4 & \leq & 2 \\ 0 & \leq & x_9 & \leq & 0 \\ -\infty & \leq & x_2 & \leq & +\infty \\ 0 & \leq & x_1 & \leq & +\infty \\ 0 & \leq & x_3 & \leq & 3 \end{array}$$

Then $\mathcal{I} = \{1, 2, 3, 4, 5, 6\}$ which decomposes, according to the types of the basic variables, into $\mathcal{I}_0 = \{3\}$, $\mathcal{I}_1 = \{2, 6\}$, $\mathcal{I}_2 = \{1, 5\}$ and $\mathcal{I}_3 = \{4\}$.

If $\mathbf{x}_{\mathcal{B}} = [1, 2, -6, -4, -2, 5]^T$ then the sets of infeasible positions are $\mathcal{M} = \{3, 5\}$ and $\mathcal{P} = \{6\}$. The degeneracy set is $\mathcal{D} = \{2\}$. This basic solution is infeasible as $\mathcal{M} \cup \mathcal{P} = \{3, 5, 6\} \neq \emptyset$. Furthermore, it is degenerate as $\mathcal{D} \neq \emptyset$.

9.2. Optimality conditions

Optimality conditions for the standard form of the LP, with type-2 variables only, were discussed in section 2.2.1. The idea how they were derived can be applied also to CF-1 with some modifications to take into account the presence of more types of variables.

Assume a feasible basis \mathcal{B} is known for (9.1) – (9.3). Repeating (2.26), the objective value can again be expressed as a function of nonbasic variables as

$$z = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{b} + (\mathbf{c}_{\mathcal{R}}^T - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{R}) \mathbf{x}_{\mathcal{R}}. \quad (9.13)$$

In this equation, the multiplier of nonbasic variable x_k is $c_k - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{a}_k$ which is the d_k reduced cost of x_k . With $\boldsymbol{\pi}^T = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1}$ being the simplex multiplier, $d_k = c_k - \boldsymbol{\pi}^T \mathbf{a}_k$.

Now we can investigate what happens if x_k moves away from its current value. (9.13) shows that if the displacement of x_k is denoted by t the objective function changes by td_k ,

$$z(t) = z + td_k, \quad (9.14)$$

i.e., the rate of the change is d_k . Since only feasible values of x_k are of interest, $\ell_k \leq x_k + t \leq u_k$ must hold for the displacement.

As there can be four different types of nonbasic variables in CF-1, x_k can be of any of them.

- 1 If $\text{type}(x_k) = 0$ then $x_k = 0$ is the only feasible value for it, therefore, type-0 nonbasic variables cannot be displaced. Consequently, they will never enter the basis and can be ignored during the test for optimality.
- 2 If $\text{type}(x_k) = 1$ then x_k is either at its lower bound of 0 or upper bound of u_k . In the former case its displacement can be nonnegative ($t \geq 0$) while in the latter nonpositive ($t \leq 0$).
- 3 If $\text{type}(x_k) = 2$ then x_k can move away from zero in the positive direction, $t \geq 0$.
- 4 A nonbasic type-3 variable can be at any level, though it is usually at zero. Since it is unrestricted in sign it can move in either direction.

The above possibilities for the nonbasic variables can be summarized in the following table.

Type of x_k	Value	Constraint on $x_k + t$	Feasible displacement t	
0	0	$x_k + t = 0$	$t = 0$	
1	0	$0 \leq x_k + t \leq u_k$	$0 \leq t \leq u_k$	(9.15)
1	u_k	$0 \leq x_k + t \leq u_k$	$-u_k \leq t \leq 0$	
2	0	$0 \leq x_k + t$	$t \geq 0$	
3	0	$-\infty \leq x_k + t \leq +\infty$	$-\infty \leq t \leq +\infty$	

From this table it is possible to identify the situation when no feasible displacement of the nonbasic variables exists that could potentially improve the objective function. This amounts to a sufficient condition of optimality. It is stated formally in the following theorem.

THEOREM 9.1 (PRIMAL OPTIMALITY CONDITIONS) *Given a feasible basis \mathcal{B} and an index set of nonbasic variables at upper bound \mathcal{U} for problem (9.1) – (9.3). Solution \mathbf{x} determined by \mathcal{B} and \mathcal{U} according to (9.7) is optimal if the d_k reduced costs of nonbasic variables satisfy the following conditions*

Type(x_k)	Value	d_k	Remark
0	$x_k = 0$	<i>Immaterial</i>	
1	$x_k = 0$	≥ 0	
1	$x_k = u_k$	≤ 0	$k \in \mathcal{U}$
2	$x_k = 0$	≥ 0	
3	$x_k = 0$	$= 0$	

PROOF. Combining the t feasible displacements (directions) in (9.15) with the corresponding sign of d_k in (9.16) shows that td_k is always nonnegative. It means that under these conditions there is no feasible direction emanating from the current solution where the objective function can improve. Therefore, \mathbf{x} is an optimal solution. \square

In case of a maximization problem the opposite sign of d_k applies in the optimality conditions. Namely, $d_k \leq 0$ if $x_k = 0$ and type(x_k) is 1 or 2, and $d_k \geq 0$ if $x_k = u_k$. Conditions for type-0 and type-3 variables, for obvious reasons, remain unchanged.

In practice, it is necessary to allow some slight violation of the feasibility as computed values of the variables can carry some computational error. Therefore, the finite bounds of variable x_k are extended so that

$$\ell_k - \varepsilon_f \leq x_k \leq u_k + \varepsilon_f,$$

where $\varepsilon_f > 0$ is the feasibility tolerance. Its value is a parameter that can be adjusted to the actual number representation and arithmetic of the computer. In case of IEEE double precision numbers reasonable values for ε_f are $10^{-8} \leq \varepsilon_f \leq 10^{-6}$. ε_f can also be determined as a fraction of the norm (the ‘size’) of $\mathbf{x}_{\mathcal{B}}$. This latter is a safe method to adjust the tolerance automatically to the magnitude of the components of the solution vector which makes the tolerance more meaningful. See section 9.3.4 for more details on feasibility tolerance.

9.3. Improving a non-optimal basic feasible solution

If the current BFS does not satisfy the (9.16) optimality conditions the simplex method attempts to move to a neighboring feasible basis with a better objective value. The implied requirements of the basis change are (i) the objective value gets better, (ii) the incoming variable takes a feasible value, (iii) the outgoing variable leaves the basis at a feasible level, and (iv) all basic variables remain feasible.

From section 2.2.2 it is known that the attempt may not be completely successful as the descent of z may be inhibited by a zero steplength. If it happens it is the result of degeneracy which can occur also with CF-1. If the optimality conditions are not satisfied there can be several variables with the ‘wrong sign’ of the reduced cost. From theoretical point of view, any of them can be chosen as an entering candidate. Practically, however, the total computational effort to solve the problem heavily depends on this choice. Unfortunately, it can not be seen in advance which is the best one but there are methods to identify good candidates. This issue will be addressed in greater detail in section 9.5.

At this stage, it is assumed that an improving candidate x_q , $q \in \mathcal{R}$, has been selected. Depending on its type (type-3 or else) and current state (at zero or upper bound) the improving displacement t can be positive or negative.

- 1 **The $d_q < 0$ case.** If $d_q < 0$ then the requirement for the improvement is $t \geq 0$. Assuming x_q changes to $x_q + t$ the basic variables also change as a function of t to satisfy the main equations in (9.8)

$$\mathbf{Bx}_B(t) + t\mathbf{a}_q = \mathbf{b}_U,$$

from which

$$\mathbf{x}_B(t) = \mathbf{B}^{-1}\mathbf{b}_U - t\mathbf{B}^{-1}\mathbf{a}_q. \quad (9.17)$$

Using notations $\alpha_q = \mathbf{B}^{-1}\mathbf{a}_q$, $\beta = \mathbf{B}^{-1}\mathbf{b}_U$, $\beta(t) = \mathbf{x}_B(t)$, (9.17) can be rewritten as

$$\beta(t) = \beta - t\alpha_q. \quad (9.18)$$

The i -th component of (9.18) is

$$\beta_i(t) = \beta_i - t\alpha_q^i. \quad (9.19)$$

The aim is to determine the largest t so that all $\beta_i(t)$, $i = 1, \dots, m$, and x_q remain feasible. As type-3 basic variables are always at a feasible level they are excluded from this investigation. Denoting

$v = u_B$ (v = Greek upsilon), i.e., $v_i = u_{k_i}$ for $i = 1, \dots, m$, the feasibility of the i -th basic variable is maintained as long as t is such that

$$0 \leq \beta_i - t\alpha_q^i \leq v_i \quad (9.20)$$

is satisfied.

Define the following two index sets

$$\mathcal{I}^+ = \{i : \alpha_q^i > 0, \text{ type}(\beta_i) \in \{0, 1, 2\}\}, \quad (9.21)$$

$$\mathcal{I}^- = \{i : \alpha_q^i < 0, \text{ type}(\beta_i) \in \{0, 1\}\}. \quad (9.22)$$

(9.19) shows that $\beta_i(t)$ is a decreasing function if $\alpha_q^i > 0$. In this case, the largest possible value t can take is $t_i = \beta_i/\alpha_q^i$, as it can be seen from the $\beta_i - t\alpha_q^i \geq 0$ part of (9.20). Obviously, all basic variables β_i , $i \in \mathcal{I}^+$, remain feasible if t is not greater than

$$\theta^+ = \min_{i \in \mathcal{I}^+} \{t_i\} = \min_{i \in \mathcal{I}^+} \left\{ \frac{\beta_i}{\alpha_q^i} \right\}. \quad (9.23)$$

This formula is nothing but the well known ratio test of (2.39).

On the other hand, if $\alpha_q^i < 0$ then $\beta_i(t)$ is an increasing function. The maximum value of t is now determined by $\beta_i - t\alpha_q^i \leq v_i$, from which $t_i = (\beta_i - v_i)/\alpha_q^i$. Therefore, all basic variables β_i , $i \in \mathcal{I}^-$, remain feasible if t is not greater than

$$\theta^- = \min_{i \in \mathcal{I}^-} \{t_i\} = \min_{i \in \mathcal{I}^-} \left\{ \frac{\beta_i - v_i}{\alpha_q^i} \right\}. \quad (9.24)$$

If any of the sets \mathcal{I}^+ and \mathcal{I}^- is empty the corresponding θ^+ or θ^- is defined to be $+\infty$. The displacement of x_q may be limited by its own bound u_q . Therefore, the maximum of t such that x_q and all basic variables remain feasible is determined by

$$\theta = \min\{\theta^+, \theta^-, u_q\}. \quad (9.25)$$

If the minimum in (9.25) is achieved for a basic position $p \in \mathcal{I}^+ \cup \mathcal{I}^-$ then the corresponding basic variable reaches its (lower or upper) bound. Thus, it can leave the basis at feasible level and can be replaced by x_q which then becomes the new basic variable in position p with a value of

$$\bar{x}_q = x_q + \theta. \quad (9.26)$$

If the minimum is not unique any position with the minimum ratio can be chosen. In practice, this ambiguity can be utilized to make a

computationally favorable selection (e.g., choose one with large pivot element). On the other hand, this situation leads to the creation of degenerate positions in \mathbf{x}_B . Details are given later.

If the minimum in (9.25) is u_q then x_q cannot enter the basis at feasible level because no basic variable reaches a bound. Therefore, the only possibility is to set x_q to its own (upper) bound.

Technically, the (9.23) – (9.24) ratio test can be performed in the following way. Take $\alpha_q^1, \dots, \alpha_q^m$ in this order. If $\alpha_q^i \neq 0$ compute t_i according to the following table

type(β_i)	$\alpha_q^i > 0$	$\alpha_q^i < 0$
0	$t_i = 0$	$t_i = 0$
1	$t_i = \frac{\beta_i}{\alpha_q^i}$	$t_i = \frac{\beta_i - u_i}{\alpha_q^i}$
2	$t_i = \frac{\beta_i}{\alpha_q^i}$	–
3	–	–

If \bar{t} is defined as $\bar{t} = \min\{t_i\}$ then θ of (9.25) is equivalent to $\theta = \min\{\bar{t}, u_q\}$.

If θ turns out to be $+\infty$ the solution is unbounded. Otherwise, $\theta < +\infty$ and it is the steplength in (9.18). Therefore, the basic variables change to

$$\bar{\beta} = \beta(\theta) = \beta - \theta \alpha_q. \quad (9.28)$$

From (9.14) we know that the new value of the objective function is $\bar{z} = z + \theta d_q$. The same update formula can be obtained if \bar{z} is recomputed from the new solution

$$\begin{aligned} \bar{z} &= \mathbf{c}_B^T \mathbf{x}_B(\theta) + c_q \theta \\ &= \mathbf{c}_B^T (\mathbf{B}^{-1} \mathbf{b}_U - \theta \mathbf{B}^{-1} \mathbf{a}_q) + c_q \theta \\ &= \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{b}_U + \theta (c_q - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{a}_q) \\ &= z + \theta d_q. \end{aligned} \quad (9.29)$$

If the minimum in (9.25) is achieved for a basic position $p \in \mathcal{I}^+ \cup \mathcal{I}^-$ then the corresponding basic variable reaches one of its bounds. Namely, if $p \in \mathcal{I}^+$ it is the lower bound, otherwise the upper bound. Thus, x_{k_p} leaves the basis at the appropriate bound and it is replaced by x_q which becomes basic in position p as the new β_p . The value of x_q is determined in (9.26) as $\bar{\beta}_p = \bar{x}_q = x_q + \theta$. The new basis is

$\bar{\mathcal{B}} = \mathcal{B} \setminus \{k_p\} \cup \{q\}$. If $p \in \mathcal{I}^-$ (outgoing leaves at upper bound) the ‘at-bound’ set is updated, $\bar{\mathcal{U}} = \mathcal{U} \cup \{k_p\}$. From (9.28) and (9.26) the new basic solution is

$$\bar{\beta}_i = \beta_i - \theta \alpha_q^i, \quad \text{for } i = 1, \dots, m, \quad i \neq p, \quad (9.30)$$

$$\bar{\beta}_p = x_q + \theta. \quad (9.31)$$

The new value of the outgoing variable is determined as the p -th component of $\bar{\beta}$ in (9.28) and is either 0 or u_{k_p} depending on whether $p \in \mathcal{I}^+$ or $p \in \mathcal{I}^-$.

If the minimum in (9.25) is u_q then no basic variable reaches any of its bounds. As the maximum feasible displacement of x_q is u_q , no basis change is possible. In this case x_q remains nonbasic but it moves to its upper bound of u_q . Such a step is called *bound flip*. The set of variables at upper bound changes to $\bar{\mathcal{U}} = \mathcal{U} \cup \{q\}$. The new value of the objective function is given by (9.29).

EXAMPLE 9.2 Assume, a BFS, $\beta = \mathbf{x}_{\mathcal{B}}$, a type-2 incoming variable, $x_q = 0$, $d_q = -3$, the objective value, $z = 12$, and α_q are given. Determine the ratios, the outgoing variable (if any), the value of the incoming variable, the new BFS and the new objective value.

It follows from the preliminaries that this is the $t \geq 0$ case.

i	$\text{type}(\beta_i)$	β_i	v_i	α_q^i	$\mathcal{I}^+/\mathcal{I}^-$	t_i
1	0	0	0	0	—	—
2	1	5	10	5	\mathcal{I}^+	$5/5 = 1$
3	1	2	4	-1	\mathcal{I}^-	$(2 - 4)/(-1) = 2$
4	2	6	∞	2	\mathcal{I}^+	$6/2 = 3$
5	2	2	∞	—	—	—

From the above, $\theta^+ = \min\{1, 3\} = 1$, $\theta^- = \min\{2\} = 2$ (or $\bar{t} = \min\{1, 2, 3\} = 1$). As $u_q = +\infty$, $\theta = \min\{\theta^+, \theta^-, u_q\} = 1$ and $p = 2$ (the subscript defining the minimum ratio). Note, despite degeneracy of the solution, the minimum ratio is positive. The new solution is

$$\beta(\theta) = \beta - 1 \times \alpha_q = \begin{bmatrix} 0 \\ 5 \\ 2 \\ 6 \\ 2 \end{bmatrix} - \begin{bmatrix} 0 \\ 5 \\ -1 \\ 2 \\ -3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 3 \\ 4 \\ 5 \end{bmatrix}.$$

Variable x_{k_2} has become 0 and leaves the basis. It is replaced by $x_q = 1$ as basic variable in position 2.

$$\bar{\mathcal{B}} = \mathcal{B} \setminus \{k_2\} \cup \{q\} \text{ and } \bar{\beta} = \mathbf{x}_{\bar{\mathcal{B}}} = [0, 1, 3, 4, 5]^T.$$

The new objective value from (9.14) is $\bar{z} = z(\theta) = z + \theta d_q = 12 + 1(-3) = 9$.

EXAMPLE 9.3 Assume, a BFS, $\beta = \mathbf{x}_{\mathcal{B}}$, a type-1 incoming variable, $x_q = 0$ with an upper bound of $u_q = 2$, $d_q = -4$, the objective value, $z = 3$, and α_q are given. Determine the ratios, the outgoing variable (if applies), the value of the incoming variable, the new BFS and the new objective value.

Obviously, it is again the $t \geq 0$ case.

i	$type(\beta_i)$	β_i	v_i	α_q^i	$\mathcal{I}^+/\mathcal{I}^-$	t_i
1	1	0	1	0	—	—
2	1	6	10	1	\mathcal{I}^+	$6/1 = 6$
3	1	1	4	-1	\mathcal{I}^-	$(1-4)/(-1) = 3$
4	2	8	∞	2	\mathcal{I}^+	$8/2 = 4$

Now, $\theta^+ = \min\{6, 4\} = 4$, $\theta^- = \min\{3\} = 3$ and $u_q = 2$. Therefore, $\theta = \min\{4, 3, 2\} = 2$.

$$\beta(\theta) = \beta - 2\alpha_q = \begin{bmatrix} 0 \\ 6 \\ 1 \\ 8 \end{bmatrix} - 2 \begin{bmatrix} 0 \\ 1 \\ -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ 3 \\ 4 \end{bmatrix}.$$

As $\theta = u_q$ there is no change in the basis, $\bar{\mathcal{B}} = \mathcal{B}$ and the iteration is a bound flip, $\bar{x}_q = x_q + \theta = 0 + 2 = 2$. The subscript of the incoming variable, q , is added to \mathcal{U} to make $\bar{\mathcal{U}} = \mathcal{U} \cup \{q\}$.

The new objective value is $\bar{z} = z(\theta) = z + \theta d_q = 3 + 2(-4) = -5$.

- 2 **The $d_q > 0$ case.** If an incoming candidate with $d_q > 0$ was selected then the improving feasible displacement of x_q is $t \leq 0$. It means that the reduced cost of either a type-1 variable at upper bound or a type-3 variable violated the optimality condition and was chosen to enter the basis negatively. We want to determine the largest (most negative) displacement, $t \leq 0$, such that requirements (i) through (iv) at the beginning of this section are satisfied.

Notations introduced for the $d_q < 0$ case will be used here, too, except \mathcal{I}^+ and \mathcal{I}^- which are defined as

$$\mathcal{I}^+ = \{i : \alpha_q^i > 0, \text{ type}(\beta_i) \in \{0, 1\}\}, \quad (9.32)$$

$$\mathcal{I}^- = \{i : \alpha_q^i < 0, \text{ type}(\beta_i) \in \{0, 1, 2\}\}. \quad (9.33)$$

Now the (9.20) constraints must hold with $t \leq 0$. If $\alpha_q^i < 0$ then $\beta_i(t)$ decreases as $t \rightarrow -\infty$. The smallest t for which $\beta_i(t)$, $i \in \mathcal{I}^-$, remains nonnegative is $t_i = \beta_i/\alpha_q^i$. Additionally, all basic variables in \mathcal{I}^- remain feasible if t is not smaller than

$$\theta^- = \max_{i \in \mathcal{I}^-} \{t_i\} = \max_{i \in \mathcal{I}^-} \left\{ \frac{\beta_i}{\alpha_q^i} \right\}.$$

If $\alpha_q^i > 0$ then $\beta_i(t)$ increases as $t \rightarrow -\infty$. If $i \in \mathcal{I}^+$ then $\beta_i(t)$ reaches its upper bound at $t_i = (\beta_i - v_i)/\alpha_q^i$. All basic variables β_i , $i \in \mathcal{I}^+$, remain feasible if t is not smaller than

$$\theta^+ = \max_{i \in \mathcal{I}^+} \{t_i\} = \max_{i \in \mathcal{I}^+} \left\{ \frac{\beta_i - v_i}{\alpha_q^i} \right\}.$$

If any of the sets \mathcal{I}^+ and \mathcal{I}^- is empty, the corresponding θ^+ or θ^- is defined to be $-\infty$. The displacement of x_q may be limited by its own lower bound if $\text{type}(x_q) = 1$. Therefore, the threshold value for t which ensures that all basic variables and also x_q remain feasible is defined by

$$\theta = \max\{\theta^+, \theta^-, -u_q\} \quad (9.34)$$

If $\theta = -\infty$ the solution is unbounded. Otherwise, there is a basis change or a bound flip (if $\theta = -u_q$). In the former case x_q replaces x_{k_p} to become the p -th basic variable, $\bar{\mathcal{B}} = \mathcal{B} \setminus \{k_p\} \cup \{q\}$. The new basic solution is determined by (9.30) and (9.31). The ‘at bound’ set is appended by k_p if the outgoing variable leaves at bound (i.e., if $p \in \mathcal{I}^+$) to become $\bar{\mathcal{U}} = \mathcal{U} \cup \{k_p\}$. If the incoming variable is of type-1, it is removed from \mathcal{U} such that $\bar{\mathcal{U}} = \mathcal{U} \setminus \{q\}$. Finally, if the incoming is of type-1 and the outgoing leaves at bound then $\bar{\mathcal{U}} = \mathcal{U} \setminus \{q\} \cup \{k_p\}$. If (9.34) defines a bound flip, the ‘at bound’ set changes to $\bar{\mathcal{U}} = \mathcal{U} \setminus \{q\}$ and the solution is updated according to (9.28).

It is easy to see that the two cases ($d_q < 0$ and $d_q > 0$) can be discussed together to a large extent. Namely, $-\alpha_q^i$ can be substituted for α_q^i in formulas (9.21)–(9.31) to obtain the $d_q > 0$ case. With this substitution, \mathcal{I}^+ of (9.32) changes to \mathcal{I}^- of (9.22) and \mathcal{I}^- of (9.33) changes to \mathcal{I}^+ of (9.21). In this way, of course, we obtain $-\theta$ in (9.25). Similarly, $-t_i$ will

be represented in (9.27). As a consequence, the transformation formulas (9.28)–(9.31) remain valid with θ (and not with its negation). The only practical difference occurs with the ‘at bound’ set when the selected x_q is at bound of u_q , as shown above.

EXAMPLE 9.4 Assumptions are the same as in Example 9.3 but the incoming variable (x_q) comes in from its upper bound of 2 ($u_q = 2$) while $d_q = 3$.

Now the displacement of x_q must be nonpositive. We substitute $-\alpha_q^i$ for α_q^i and use the \mathcal{I}^+ and \mathcal{I}^- sets of the $d_q < 0$ case ($t \geq 0$). We also use the ratio test defined by (9.27) which will now yield $-t_i$.

i	$type(\beta_i)$	β_i	v_i	$-\alpha_q^i$	$\mathcal{I}^+/\mathcal{I}^-$	$-t_i$
1	0	0	0	0	—	—
2	1	6	10	-1	\mathcal{I}^-	$(6 - 10)/(-1) = 4$
3	1	1	4	1	\mathcal{I}^+	$1/1 = 1$
4	2	8	∞	-2	—	—

From the table, $-\bar{t} = \min\{4, 1\} = 1$ and $p = 3$. Now $-\theta = \min\{-\bar{t}, u_q\} = \min\{1, 2\} = 1$, giving $\theta = -1$.

$$\boldsymbol{\beta}(\theta) = \boldsymbol{\beta} - (-1)\boldsymbol{\alpha}_q = \begin{bmatrix} 0 \\ 6 \\ 1 \\ 8 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 7 \\ 0 \\ 10 \end{bmatrix}.$$

Variable x_{k_3} has become 0 and leaves the basis. It is replaced by x_q with its new value of $\bar{x}_q = x_q + \theta = 2 + (-1) = 1$.

The new basis and basic feasible solution are $\bar{\mathcal{B}} = \mathcal{B} \setminus \{B_3\} \cup \{q\}$ and $\mathbf{x}_{\bar{\mathcal{B}}} = [0, 7, 1, 10]^T$.

9.3.1 The logic of the ratio test

For easier reference, we summarize the logic of the ratio test developed for CF-1 of the LP problem in a formalized procedure called Algorithm 1. It is assumed the reduced cost d_q of a nonbasic variable x_q failed the (9.16) optimality test and was chosen to enter the basis. It is also assumed the infinity is represented numerically by a large number that does not cause overflow when it takes part in comparisons. In this way θ is initialized to the maximum distance the incoming variable can travel, which is (in absolute value) equal to u_q ($+\infty$ allowed).

The description of Algorithm 1 which is given in pseudo code (page 174) reflects just the logic. Computational considerations are discussed in section 9.3.4.

Algorithm 1 Primal Phase-2 Ratio Test

Require: A feasible basis with \mathcal{B} , \mathcal{U} and $\mathbf{x}_{\mathcal{B}}$, entering candidate x_q with upper bound u_q , reduced cost d_q and updated column α_q .

Ensure: Steplength θ , Pivot row p . If $|\theta| = +\infty$ then solution is unbounded.

```

1: Initialize:  $k := p := r := 0$ ;  $\theta := u_q$ 
2:  $\sigma := 1$  {Signalling the  $t \geq 0$  case}
3: if  $d_q > 0$  then
4:    $\sigma := -1$  {Signalling  $t \leq 0$ }
5: end if
6: for  $i := 1$  to  $m$  do
7:   if  $\alpha_q^i \neq 0 \wedge \text{type}(x_{k_i}) \neq 3$  then {type-3 basic variables ignored}
8:      $a := \sigma \times \alpha_q^i$ 
9:     if  $a > 0$  then
10:        $t := x_{k_i}/a$ 
11:        $k := 1$  { $i$ -th basic variable reaches lower bound}
12:     else
13:        $t := (x_{k_i} - u_{k_i})/a$ 
14:        $k := -1$  { $i$ -th basic variable reaches upper bound}
15:     end if
16:     if  $t < \theta$  then
17:        $\theta := t$  {Minimum ratio so far}
18:        $p := i$  {Row index of minimum ratio}
19:        $r := k$  {Way of leaving the basis (UB or LB)}
20:     end if
21:   end if
22: end for
23: if  $p = 0$  then {No ratio was defined}
24:   if  $u_q < +\infty$  then {Upper bound of  $x_q$  is finite}
25:      $\theta := u_q$  {Bound flip}
26:   else
27:     Solution unbounded.
28:   end if
29: end if
30:  $\theta := \sigma \times \theta$  {Restore proper sign for  $\theta$ }
```

9.3.2 Extensions to the ratio test

In CF-2 minus type (type-4) variables may also be present. In this section we show how the ratio test developed for CF-1 can be extended to deal with type-4 variables. Additionally, we point out how to handle the case when one of the finite bounds of a variable is not moved to zero but kept at its original nonzero value. It is usually the lower bound but in case of a type-4 variable it can be the upper bound.

Type-4 variables can be basic and/or nonbasic. We investigate them in both situations.

If there is a type-4 variable in a feasible basis, say in position i , then $-\infty \leq \beta_i \leq 0$ holds. It restricts the t displacement of the incoming variable by $-\infty \leq \beta_i - t\alpha_q^i \leq 0$. Therefore, if $t \geq 0$ is required then this position imposes a restriction only if $\alpha_q^i < 0$, in which case a ratio $t_i = \beta_i/\alpha_q^i \geq 0$ is defined. In the opposite case ($t \leq 0$), a ratio is defined if $\alpha_q^i > 0$ resulting in a nonpositive $t_i = \beta_i/\alpha_q^i$.

The optimality condition for a type-4 nonbasic variable can be derived from (9.13) and (9.15) and is

$$d_k \leq 0, \text{ type}(x_k) = 4.$$

If such a nonbasic variable, say x_q , violates this condition and is chosen to enter the basis then $d_q > 0$. Therefore, the displacement of x_q must be nonpositive which is the $t \leq 0$ case. Consequently, the appropriate procedure developed for $t \leq 0$ (including the presence of type-4 basic variables) can be used.

As indicated in section 1.3.1, the translation of finite lower or upper bounds may not be desirable. Namely, some algorithmic techniques require these values explicitly. For such cases the rules of the ratio test need to be amended. The key idea is unchanged (see the first paragraph of section 9.3). For notational simplicity, the lower bound vector of basic variables will be denoted by $\lambda = \ell_B$, so that $\lambda_i = \ell_{k_i}$, and notation $v = u_B$ introduced in section 9.3 will also be used. The t displacement of the incoming variable must be such that all basic variables remain feasible, i.e.,

$$\lambda_i \leq \beta_i - t\alpha_q^i \leq v_i. \quad (9.35)$$

For the $t \geq 0$ case (9.35) says that the i -th basic variable reaches its finite lower bound if $t_i = (\beta_i - v_i)/\alpha_q^i$ for $\alpha_q^i < 0$ or its finite upper bound if $t_i = (\beta_i - \lambda_i)/\alpha_q^i$ for $\alpha_q^i > 0$. Therefore, the equivalent of the

(9.27) ratio table is

type(β_i)	$\alpha_q^i > 0$	$\alpha_q^i < 0$	
0	$t_i = 0$	$t_i = 0$	
1	$t_i = \frac{\beta_i - \lambda_i}{\alpha_q^i}$	$t_i = \frac{\beta_i - v_i}{\alpha_q^i}$	
2	$t_i = \frac{\beta_i - \lambda_i}{\alpha_q^i}$	—	
3	—	—	
4	—	$t_i = \frac{\beta_i - v_i}{\alpha_q^i}$	

(9.36)

Having determined the t_i ratios the rest of the algorithm works exactly the same way as described for the $t \geq 0$ case in CF-1. If the displacement of the incoming variable is negative then α_q^i is to be replaced by $-\alpha_q^i$ and the rules of the $t \geq 0$ case apply as pointed out after the $d_q > 0$ case in section 9.3. Recall, by this substitution $-\theta$ is determined.

For completeness and easier reference, the optimality conditions that cover all cases are also restated. They differ from the ones in (9.16) in two ways, namely, type-4 variables and also original bounds are explicitly included.

Type(x_k)	Value	d_k	Remark
0	$x_k = \ell_k$	Immaterial	
1	$x_k = \ell_k$	≥ 0	
1	$x_k = u_k$	≤ 0	$k \in \mathcal{U}$
2	$x_k = \ell_k$	≥ 0	
3	$x_k = 0$	$= 0$	
4	$x_k = u_k$	≤ 0	

(9.37)

Appending the logical description of the primal phase-2 ratio test, Algorithm 1, that corresponds to (9.36) is a straightforward exercise and is left to the reader.

Note, a type-3 nonbasic variable can have a nonzero value. However, the optimality condition for such a variable is still $d_k = 0$ since it is the feasible displacement of this variable that needs to be ‘non-profitable’ in (9.37) (which holds only if $d_k = 0$).

9.3.3 Algorithmic description of PSM-G

Based on the developments in the preceding subsections it is possible to give a step-by-step algorithmic description of the simplex method for

the LP problem of the form

$$\begin{aligned} \text{minimize } & z = \mathbf{c}^T \mathbf{x} \\ \text{subject to } & \mathbf{Ax} = \mathbf{b}, \\ & \text{and type restrictions on the variables.} \end{aligned} \tag{9.38}$$

if a feasible basis $\mathcal{B} = \{k_1, \dots, k_m\}$ is known together with the \mathcal{U} index set of nonbasic variables at upper bound. In the description we define some operations with \mathbf{B}^{-1} . They can be replaced by operations with the LU form.

If a feasible basis is not available we need to find one. The way to do it for CF-1 (or CF-2) is discussed in section 9.6.

Primal Simplex Method #1 (PSM-G)

Step 0. Initialization. Verify that the initial basis \mathcal{B} with \mathcal{U} is feasible.

Determine \mathbf{B}^{-1} , $\beta = \mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}\mathbf{b}_{\mathcal{U}}$ and $z = \mathbf{c}_{\mathcal{B}}^T \mathbf{x}_{\mathcal{B}} + \mathbf{c}_{\mathcal{U}}^T \mathbf{x}_{\mathcal{U}}$.

Step 1. Compute simplex multiplier $\pi^T = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1}$.

Step 2. Pricing. (PRICE) Compute reduced costs $d_j = c_j - \pi^T \mathbf{a}_j$, for $j \in \mathcal{R}$ and check the (9.16) optimality conditions. If they are satisfied exit with “optimal solution found”. Otherwise, there is at least one variable with $d_j < 0$ or $d_j > 0$ that violates the optimality conditions. Usually there are several. Choose one from among the candidates using some (simple or complex) criterion. The subscript of the selected variable is denoted by q .

Step 3. Transform the column vector \mathbf{a}_q of the improving candidate: $\alpha_q = \mathbf{B}^{-1} \mathbf{a}_q$.

Step 4. Choose row. (CHUZRO) Perform ratio test using Algorithm 1 or its extended version as appropriate to obtain θ . If $\theta = +\infty$ conclude that “solution is unbounded” and exit.

Step 5. Update:

Objective value: $\bar{z} = z(\theta) = z + \theta d_q$.

If the iteration is a bound flip add or remove q to/from \mathcal{U} depending on the direction of the displacement (determined by the sign of d_q). Update solution according to (9.28): $\bar{\beta} = \beta - \theta \alpha_q$.

Otherwise the iteration involves a basis change which entails further updating.

- Solution, according to (9.28) and (9.26):

$$\begin{aligned}\bar{\beta}_i &= \beta_i - \theta \alpha_q^i, \quad \text{for } i = 1, \dots, m, i \neq p, \\ \bar{\beta}_p &= x_q + \theta.\end{aligned}$$

- Basis inverse in product form or the LU factorization (depending on which is being used).
- Upper bound list:
if the incoming variable comes in from upper bound then $\bar{\mathcal{U}} = \mathcal{U} \setminus \{q\}$,
if the outgoing leaves at upper bound then $\bar{\mathcal{U}} = \mathcal{U} \cup \{k_p\}$.
- Basis: x_q becomes basic in position p : $\bar{\mathcal{B}} = \mathcal{B} \setminus \{k_p\} \cup \{q\}$.

Return to Step 1 with quantities with bar $\bar{\cdot}$ like $\bar{\mathcal{B}}$ replacing their respective originals, like \mathcal{B} .

9.3.4 Computational considerations

When implementing the ratio test the following observations are useful to remember.

- 1 The $t \geq 0$ and the $t \leq 0$ cases can be handled by the same algorithm (as already done in Algorithm 1). One can be turned into the other if α_q^i is replaced by $-\alpha_q^i$ in the tests.
- 2 If the minimum is not unique in the ratio test then any of the basic variables that is involved in the tie can be chosen to leave the basis. This freedom can be exploited to choose the largest pivot element among the tied rows. As the other variables in the tie will also go to their bounds the new basic solution will become degenerate.

The theoretical possibility of equal ratios is not completely realized in practice because of the emerging (usually small) computational errors. Therefore we will experience nearly equal ratios and ‘near degeneracy’, i.e., basic variables very close to their bounds. Actually, the small errors act as perturbations making the smallest ratio nearly always unique. However, relying on this resolution of degeneracy is not very effective in practice. See section 9.7 for a more complete discussion of coping with degeneracy.

- 3 As in practice the feasibility of the basic variables is satisfied with a tolerance the ratio test has to be modified accordingly. The appropriate formulas can be derived from (9.20) by using the extended bounds

$$-\varepsilon_f \leq \beta_i - t \alpha_q^i \leq v_i + \varepsilon_f,$$

where $\varepsilon_f > 0$ is the feasibility tolerance. As a result, for example, the line corresponding to type-1 variables in (9.27) becomes

$$t_i = \frac{\beta_i + \varepsilon_f}{\alpha_q^i} \quad t_i = \frac{\beta_i - v_i - \varepsilon_f}{\alpha_q^i}. \quad (9.39)$$

For future reference, we call the first fraction *type-A ratio* and the second one *type-B ratio*. The $\varepsilon_f > 0$ feasibility tolerance can be determined as a fraction of $\|\mathbf{x}_B\|$ or a uniform value throughout the iteration. In the latter case a typical values are $10^{-8} \leq \varepsilon_f \leq 10^{-6}$ if standard double precision arithmetic with 53 bit mantissa is used.

Type-0 basic variables are not included in this approach for the following reason. It is advantageous to get rid of type-0 basic variables because such a variable x_{k_i} will block the displacement whenever $\alpha_q^i \neq 0$ holds. Therefore, letting these variables define 0 ratios and choosing one of them to leave the basis increases the chances of future nondegenerate iterations.

4 Harris's ratio test.

The use of feasibility tolerance in determining the outgoing variable can be utilized for finding better sized pivot elements which is crucial for the numerical stability of the simplex method. Harris [Harris, 1973] was the first to propose a method that serves this purpose. Any serious implementation must include some method of this type.

Harris's main idea is to compute the minimum ratio, say θ_1 , with a tolerance using formulas in (9.39). Next, set the tolerance to zero and recompute the eligible ratios. This time the ratios will be smaller as feasibility tolerance is not used. Check the pivot element of each position which defines a ratio not greater than θ_1 and include them in a set \mathcal{T} . Finally, choose a member from \mathcal{T} for which the pivot element is the largest in magnitude. This procedure requires two passes of the pivot column. Formally, it can be described as follows.

- (a) Determine the eligible ratios using feasibility tolerance as in (9.39) with the additional condition that $|\alpha_q^i| > \varepsilon_p$, where ε_p is the pivot tolerance. ε_p can be determined as a fraction of $\|\alpha_q\|$ or some predefined value. In the latter case a typical value is around 10^{-5} if double precision arithmetic is used.

Let

$$\hat{\varepsilon}_f = \begin{cases} 0, & \text{if } \text{type}(\beta_i) = 0, \\ \varepsilon_f & \text{otherwise.} \end{cases}$$

For the $t \geq 0$ case we have

$$\hat{t}_i = \begin{cases} \frac{\beta_i + \hat{\varepsilon}_f}{\alpha_q^i}, & \text{if } \alpha_q^i > \varepsilon_p, \\ \frac{\beta_i - v_i - \hat{\varepsilon}_f}{\alpha_q^i}, & \text{if } \alpha_q^i < -\varepsilon_p. \end{cases}$$

Let $\theta_1 = \min\{\hat{t}_i\}$. Assume the incoming variable is not blocked by its own upper bound, i.e., $\theta_1 \leq u_q$.

- (b) Compute the ratios again, this time with the exact bounds by setting $\hat{\varepsilon}_f$ to zero.

$$t_i = \begin{cases} \frac{\beta_i}{\alpha_q^i}, & \text{if } \alpha_q^i > \varepsilon_p, \\ \frac{\beta_i - v_i}{\alpha_q^i}, & \text{if } \alpha_q^i < -\varepsilon_p. \end{cases}$$

Put the row indices of ratios smaller than θ_1 into \mathcal{T} , i.e.,

$$\mathcal{T} = \{i : t_i \leq \theta_1\}.$$

- (c) Choose $p \in \mathcal{T}$ such that $|\alpha_q^p| \geq |\alpha_q^i| \forall i \in \mathcal{T}$.

$\theta = t_p$ is the steplength and p is the pivot row, while variable x_{k_p} leaves the basis at lower or upper bound depending on the type of the defining ratio (at lower bound if type-A and upper if type-B).

It can happen that β_p was slightly infeasible (within the tolerance) at the beginning of the iteration which makes θ negative. In this case, to avoid a deteriorating step, θ is set to zero. After the transformation the basic variables remain feasible within the tolerance.

The primal steplength θ is sometimes denoted by θ_P to distinguish it from the steplength in the dual (where it is denoted by θ_D).

The second pass of this procedure can be made faster if, during the first one, a list of eligible positions is built up together with the type of the ratio (type-A or type-B).

The above is a nearly perfect replica of Harris's ratio test. The only important difference is the exclusion of the type-0 basic positions from the first pass for reasons explained at the end of the previous section that discussed the need for using tolerances and getting rid of type-0 basic variables. In some numerically very difficult cases, exceptionally, the tolerances can be applied to type-0 variables, too.

When a small negative θ is replaced by zero then a small error is made as the outgoing variable is artificially placed to its exact bound. The error, is the distance of the current value of the variable from its nearest bound, $e = |\beta_i|$ or $e = |\beta_i - v_i|$, respectively. Thus $e \leq \varepsilon_f$ holds. This feasibility error may cause an error in the order of ε_f in satisfying $\mathbf{Bx}_B + \mathbf{Rx}_r = \mathbf{b}$ which is much larger than the machine precision. If this sort of infeasibility does occur it has to be eliminated by—sometimes many—costly simplex iterations. This observation has lead Gill et al. [Gill et al., 1989] to accept nonbasic variables with values different from their bounds. Their EXPAND procedure is discussed in section 9.7.3.

The introduction of the Harris ratio test has greatly improved the numerical stability of the simplex method. It has also opened up a stream of new ideas to cope with degeneracy. Therefore, in any powerful implementation of the simplex method this or some other similar method must be included. A possible refinement of this ratio test is discussed in section 9.7. It is worth reading it as it can help decide which version to implement.

- 5 It is obvious that a bound flip is a more advantageous iteration than a basis change. First of all, in this case there is a positive progress towards optimality (nondegenerate step). Furthermore, as there is no change in the basis, the previous simplex multiplier and all the reduced costs remain unchanged. This leads to a considerable saving for the next iteration. Unfortunately, it is not possible to predict which type-1 variables would lead to bound flips. This is determined by the ratio test which is performed only on the already selected column.

9.4. Determining the reduced costs

Reduced costs play a crucial role in the simplex method. They were defined in (2.27) and (2.29) and are repeated here for easier reference.

$$d_j = c_j - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{a}_j = c_j - \boldsymbol{\pi}^T \mathbf{a}_j, \quad (9.40)$$

where $\boldsymbol{\pi}$ is the simplex multiplier defined in (2.28). The optimality of a basis is expressed in terms of the d_j -s of the nonbasic variables as shown in (9.16). Furthermore, if a basis is nonoptimal the reduced costs indicate the improving search directions.

There are several ways the reduced costs belonging to a given basis can be determined as they can be computed or updated after every iteration. In this section we discuss some computationally efficient methods for obtaining the reduced costs.

9.4.1 Computing d_j

The first, and most obvious, choice is to recompute the d_j -s from their definition using the simplex multiplier π in the following way. Assume \mathbf{B} is the current basis. Determine the simplex multiplier, $\pi^T = \mathbf{c}_B^T \mathbf{B}^{-1}$, and compute the reduced costs $d_j = c_j - \pi^T \mathbf{a}_j$.

Computationally, this procedure requires the BTRAN of \mathbf{c}_B and a dot product with the nonbasic columns. Depending on the density of \mathbf{c}_B this operation can be quite demanding as the computational effort of BTRAN is proportional to the number of nonzeros in the vector to be BTRAN-ed. Whether to compute the reduced costs for all nonbasic variables or just for a fraction of them depends on the *pricing strategy* to be discussed in section 9.5.

The great advantage of recomputing d_j -s is its independence of the actual pricing strategy. This is particularly useful when different strategies are used during a solution. Additionally, at the end a complete recomputation is needed to verify optimality. Therefore, determining π is an important functionality of an advanced implementation of the simplex method and must be included in the computational toolbox.

9.4.2 Updating π

Assume the current basis is \mathbf{B} and a basis change took place replacing x_{k_p} by x_q , where x_q was chosen because of its profitable reduced cost d_q . The basis after the change is $\bar{\mathbf{B}}$. The new simplex multiplier $\bar{\pi}$ can be determined using the inverse of $\bar{\mathbf{B}}$.

Another possibility is to update the simplex multiplier as pointed out by Tomlin [Tomlin, 1974]. From (2.76), the reduced costs of the nonbasic variables (except that of the currently leaving variable x_{k_p}) transform as

$$\bar{d}_j = d_j - d_q \frac{\alpha_j^p}{\alpha_q^p}, \quad (9.41)$$

which can also be obtained from its definition as

$$\bar{d}_j = c_j - \bar{\pi}^T \mathbf{a}_j. \quad (9.42)$$

Let ρ^p denote the p -th row of \mathbf{B}^{-1} , which is in the new basis

$$\bar{\rho}^p = \mathbf{e}_p^T \bar{\mathbf{B}}^{-1}. \quad (9.43)$$

Equation (9.41) can be rewritten as

$$\bar{d}_j = d_j - d_q \frac{\rho^p \mathbf{a}_j}{\alpha_q^p} = d_j - d_q \bar{\rho}^p \mathbf{a}_j \quad (9.44)$$

because row p of the basis inverse gets transformed by dividing it by the pivot element α_q^p . Substituting d_j from (9.40) into (9.44) we obtain

$$\begin{aligned}\bar{d}_j &= c_j - \boldsymbol{\pi}^T \mathbf{a}_j - d_q \bar{\rho}^p \mathbf{a}_j \\ &= c_j - (\boldsymbol{\pi}^T - d_q \bar{\rho}^p) \mathbf{a}_j.\end{aligned}\quad (9.45)$$

Comparing (9.42) and (9.45) we can conclude that

$$\bar{\boldsymbol{\pi}}^T = \boldsymbol{\pi}^T - d_q \bar{\rho}^p. \quad (9.46)$$

Therefore, to update the simplex multiplier, we need to determine row p of the inverse of the new basis, as defined in (9.43). As it requires the BTRAN of a unit vector we can expect that this computation can be done with substantially less work than the BTRAN of the new, and occasionally quite dense, $\bar{\mathbf{c}}_{\mathcal{B}}$.

The way to use the update formula is the following. Save d_q at the end of the current iteration. Determine $\bar{\rho}^p$ from (9.43) as the first step of the next iteration then compute $\bar{\boldsymbol{\pi}}$ by (9.46). Finally, the new $\bar{\boldsymbol{\pi}}$ can be used to calculate \bar{d}_j in the pricing step of the next iteration.

The computational saving itself may justify the use of the (9.46) update formula. However, it has several other potential uses, discussed later, which just magnify the importance of it.

9.4.3 Updating d_j

The other ‘extreme’ in determining the reduced costs is complete updating. It requires the maintenance of the row vector of updated d_j -s.

The algebraic equation of updating d_j is shown in (9.41). The α_j^p/α_q^p part of it is equal to $\bar{\alpha}_j^p$ as the pivot row is transformed by dividing every element in it by α_q^p . On the other hand,

$$\bar{\alpha}_j^p = \mathbf{e}_p^T \bar{\alpha}_j = \mathbf{e}_p^T \bar{\mathbf{B}}^{-1} \mathbf{a}_j = \bar{\rho}^p \mathbf{a}_j. \quad (9.47)$$

Therefore,

$$\bar{d}_j = d_j - d_q \bar{\rho}^p \mathbf{a}_j. \quad (9.48)$$

The computational work involved in determining \bar{d}_j from (9.48) consists of computing $\bar{\rho}^p$ from (9.43) and a dot product with the nonbasic columns except k_p . The latter is simply equal to $-d_q/\alpha_q^p$, see (2.76).

It is typical that $\bar{\rho}^p$ is sparse. It means we have dot products of two sparse vectors in (9.48). (9.41) suggests that no update is needed for columns with $\bar{\alpha}_j^p = 0$. Unfortunately, these values are not known. Thus, this potential saving cannot materialize. However, if we consider the vector form of (9.48),

$$\bar{\mathbf{d}}_{\mathcal{R}}^T = \mathbf{d}_{\mathcal{R}}^T - d_q \bar{\rho}^p \mathbf{R}$$

then the following can be done. Assume there is enough space to store \mathbf{A} not only columnwise but also rowwise. Now

$$\mathbf{R} = \begin{bmatrix} \mathbf{r}^1 \\ \vdots \\ \mathbf{r}^i \\ \vdots \\ \mathbf{r}^m \end{bmatrix},$$

where \mathbf{r}^i denotes the subvector of \mathbf{a}^i (row i of \mathbf{A}) containing its nonbasic components. In this way the $\bar{\rho}^p \mathbf{R}$ product can be written as

$$\bar{\rho}^p \mathbf{R} = [\bar{\rho}_1^p, \dots, \bar{\rho}_i^p, \dots, \bar{\rho}_m^p] \begin{bmatrix} \mathbf{r}^1 \\ \vdots \\ \mathbf{r}^i \\ \vdots \\ \mathbf{r}^m \end{bmatrix} = \sum_{i=1}^m \bar{\rho}_i^p \mathbf{r}^i. \quad (9.49)$$

In other words, if $\bar{\rho}^p \mathbf{R}$ is computed according to (9.49) then the only rows that are involved in the updating of the reduced costs are the ones that correspond to nonzero coefficient of $\bar{\rho}^p$. As $\bar{\rho}^p$ can be very sparse (very often just few nonzeros), this way of updating $\mathbf{d}_{\mathcal{R}}$ can be computationally very efficient.

It should also be remembered that (9.49) is practically the computation of the updated pivot row with the basis after the current iteration. The presented method can be used more widely than just for updating the reduced costs. Further important application will be discussed at the relevant places.

The conclusion is that if there is enough memory available then this method is unbeatable in speed in case of sparse problems. Therefore, an advanced implementation should include it in the toolbox and activate it whenever the conditions are met (sparse problem, availability of memory).

9.5. Selecting an incoming (improving) variable

In the general step of phase-2 of the simplex method an improving nonbasic variable is chosen to enter the basis. If none can be found the current basic feasible solution is optimal. The selection is made by checking the (9.16) optimality conditions for each nonbasic variable. This operation is often referred to as *pricing*. Any variable that violates its optimality condition is an improving candidate. This is an embarrass-

ing freedom that has given rise to several pricing strategies. Currently none of them is known to be the best one.

From practical point of view, the best pricing strategy is the one that leads to an optimal solution with the least computational effort (which usually translates into the shortest solution time). This would be a global strategy. Unfortunately, at any specific iteration the simplex method can see only the neighboring bases of the current basic solution. As such, we cannot expect more than a locally good selection.

It is typical that a large proportion of the total solution time is spent in pricing. Therefore, it is essential to economize this operation.

Regarding the strategies, two extremes can be identified. Trying to minimize the number of iterations results in very slow iterations involving heavy computations per iteration. On the other extreme, methods that have high iteration speed usually require a very large number of iterations until termination. There is no clear theoretical answer which is better but experience shows that usually it is worth making some extra effort to improve the quality of the choice.

In this section we look at several pricing strategies and discuss their advantages and weaknesses. We also point out the need for implementing more than one strategy and making it possible to combine them if it appears beneficial.

There are several ways to categorize the pricing methods. First of all, there can be an explicitly maintained row vector containing the reduced costs being updated after every basis change. On the opposite end, the reduced costs may be computed from their definition of (2.27) in every iteration. It is not immediately obvious which of the two is more efficient because there are many other factors that can influence the efficiency.

Another aspect is whether we check all nonbasic reduced costs, called *full pricing*, or only a subset of them if a sufficient number of improving candidates have already been encountered, called *partial pricing*.

One more aspect is how the profitable (potentially improving) reduced costs are compared. If the computed d_j -s are directly compared then we talk about *direct pricing*. If they are (perhaps approximately) brought to some common platform for comparison then it is termed *normalized pricing*.

Furthermore, there are special purpose methods in pricing when the column selection is aiming at some particular goal in addition to choosing an improving candidate. In such cases we use multiple selection criteria (the first always being the profitability). This is facilitated by the use of additional price vectors (similarly formed as the simplex multiplier π).

All the above pricing ideas select one single candidate. At early stages of computer technology when resources were more restricted matrix A

was often stored on a slow background memory (disk or even magnetic tape). In such a case, checking all nonbasic reduced costs required bringing the entire matrix into the main memory at every iteration. This slowed down the iterations considerably. To—at least partly—overcome this problem the following idea has emerged. Select some of the most promising candidates during a pricing pass and do iterations with the current basic variables and the selected few. This method is referred to as *multiple pricing*. The selection of candidates is called a *major iteration* (actually, no iteration but selection is done) and the subsequent iterations are *minor iterations*. If the selected subproblem is solved entirely we talk about *complete suboptimization*, if the process terminates when none of the selected candidates can be entered with a positive gain we call it *partial suboptimization*.

Multiple pricing has several attractive features.

First of all, from the updated few columns we can select the one that makes the largest improvement in the objective function. To be more specific, assume $H > 1$ candidates have been selected and, for notational simplicity, they are $\mathbf{a}_1, \dots, \mathbf{a}_H$. After updating they become $\alpha_1, \dots, \alpha_H$. If we perform a ratio test on each of them we obtain the θ_h displacement for $h = 1, \dots, H$. This enables us to compute the attainable progress $\theta_h d_h$ (see (2.44)) and choose the candidate with the largest gain. In the updating step of the simplex method not only the solution and basis inverse but also the unused α_j candidates are updated. This procedure is repeated as long as there remains an improving candidate according to the rules of the complete or partial suboptimization. In this way, the minor iterations are computationally cheap as a new candidate can be obtained by a simple update. Of course, it can happen that after the first candidate entered the basis all the waiting ones lose their profitability which means they were updated in vain. If, however, sufficiently unrelated candidates have been selected there is a good chance that most of them can enter the basis. This issue will be further discussed later. H is usually a parameter that can take values up to 10. Too large H can result in too much extra computations. Additionally, during minor iterations the global aspects of the total neighborhood are not seen which can lead to insignificant ‘small’ iterations. Therefore, it is practical to abandon minor iterations if the improvement falls below a dynamically determined threshold. Typical values for H are $4 \leq H \leq 6$. Multiple pricing is meant to be successful if more than half of the selected candidates actually enter the basis, on the average.

Second, multiple pricing can be effective in coping with degeneracy. Simply, if any of the selected candidates can result in a positive step ahead, it will automatically be chosen.

Third, if the pivot element in the chosen column is not of proper size, the remaining columns can be searched to find a better one. And it comes at no additional cost.

Though multiple pricing does not fit into all pricing strategies to be discussed it is still an important technique and worth including in an advanced solver.

9.5.1 Dantzig pricing

In his seminal book on the simplex method [Dantzig, 1963], Dantzig suggested the selection of the most violating nonbasic variable for entering the basis. The motivation for it is that the d_j is the rate of change of the objective function as x_j is displaced from its current value.

While the idea is straightforward it does not take into account the ensuing movement of the basic variables which becomes known only during the ratio test. It is not uncommon that all the nonzero entries in the updated column α_q are large if the reduced cost is large. As some of these values appear in the denominator of the ratio test they will cause small ratios, thus a small θ_q displacement of x_q . Therefore, the advantage of a large reduced cost can easily disappear, as it is the $\theta_q d_q$ product that determines the actual gain of introducing a_q into the basis.

While this selection method was proposed as a full pricing it can also be used in other pricing frameworks. Despite its obvious potential drawback the Dantzig pricing strategy is still widely used, mostly in combination with others, in particular with multiple pricing.

9.5.2 Partial pricing

If there are many improving candidates a good selection of them can be found by scanning only a part of the nonbasic variables. This practice is known as *partial pricing*. The idea creates the possibility of different tactics how to partition the matrix for pricing. If the partitions are determined once we talk about *static partitioning*. The other possibility is to redefine the partitions as optimization progresses which is called *dynamic partitioning*. In either case, pricing starts in the partition next to the one where it stopped during the previous iteration. This goes on in a wrap around fashion after the last partition is scanned. Within partitions it is possible to do full or partial pricing. The latter is a sort of *nested partial pricing*.

If there are too few or no candidates in the currently scanned partition pricing moves on to the next one. It is a good practice to continue

scanning until a certain number of improving candidates have been encountered to give a potentially better list for multiple pricing.

It is essential to scan the entire matrix to be able to declare optimality.

A possible template for a *dynamic partial pricing* is the following.

- 1 Do full pricing after each reinversion of the basis, count the profitable nonbasic variables and denote that number by T .
- 2 In the subsequent (major) iterations: do scanning until ν improving candidates have been encountered where ν is defined to be the maximum of some fraction of m (number of rows), some fraction of T , and H (the number of vectors in suboptimization. A realistic choice is $\nu = \max\{m/10, T/4, H\}$.

If $\nu > H$ a new candidate is compared with the weakest on the list and a replacement takes place if the newcomer is more profitable.

9.5.3 Sectional pricing

Real life LP problems are often very large in size and usually have some structure. This is because they typically represent multi-period, multi-location, dynamic or stochastic models. In such cases the nonzeros of matrix \mathbf{A} are located in clusters. The projection of the clusters on the horizontal axis forms sections. Sections are usually consecutive (unless the columns have been reshuffled for some reasons). Pointers can be set up to mark the boundaries of the sections. If partial pricing is done along the sections we talk about *sectional pricing*. This concept has a deeper meaning than just being a partial pricing.

It is typical that vectors in a section are more related to each other than to vectors in other sections. Therefore, if multiple pricing is used we can obtain unrelated (or loosely related) vectors if they are picked from different sections which can lead to more meaningful minor iterations. Namely, in this case we can expect that the inclusion of one of the candidates in the basis will not deteriorate the profitability of the remaining ones and they also enter the basis.

The idea can further be refined. When building up the list of H candidates, if the list is full a newcomer is compared with the weakest from its own section if there is a representative from the section already on the list.

More variations emerge if we do partial pricing within the sections and also do partial pricing with the sections themselves. It means that only a given number of sections are scanned per major iteration and partial pricing is done within the sections. It requires a carefully designed structure to cover all possibilities including the cases when some sectors

have no improving candidates, or we are at an optimal solution. In subsection 9.5.5 we give an algorithmic framework that covers several pricing tactics including variations of sectional pricing.

9.5.4 Normalized pricing

The reduced cost of variable x_j has been defined to be $d_j = c_j - \boldsymbol{\pi}^T \mathbf{a}_j$. It can easily be seen that the actual value of d_j depends on the column scale. Namely, if $\hat{\mathbf{a}}_j = s_j \mathbf{a}_j$ and $\hat{c}_j = s_j c_j$, where $s_j > 0$ is the scale factor, then

$$\hat{d}_j = \hat{c}_j - \boldsymbol{\pi}^T \hat{\mathbf{a}}_j = s_j(c_j - \boldsymbol{\pi}^T \mathbf{a}_j) = s_j d_j.$$

The unfortunate consequence of this observation is that if the magnitude of the reduced cost is used in comparison we may get a misleading result. However, we can obtain a scale independent measure for d_j if we apply the following *dynamic scaling*:

$$\tilde{d}_j = \frac{d_j}{\|\boldsymbol{\alpha}_j\|}, \quad (9.50)$$

where $\|\cdot\|$ denotes the Euclidean norm. To see the scale independence of \tilde{d}_j , we note that $\hat{\boldsymbol{\alpha}}_j = \mathbf{B}^{-1} \hat{\mathbf{a}}_j = s_j \mathbf{B}^{-1} \mathbf{a}_j = s_j \boldsymbol{\alpha}_j$, and, therefore,

$$\|\hat{\boldsymbol{\alpha}}_j\| = s_j \|\boldsymbol{\alpha}_j\|.$$

So, if the scaled column $\hat{\mathbf{a}}_j$ is used instead \mathbf{a}_j then both the numerator and denominator are multiplied by s_j in (9.50). Therefore, \tilde{d}_j remains unchanged. Column selection based on (9.50) is called *steepest edge pricing*.

Greenberg and Kalan [Greenberg and Kalan, 1975] give a further motivation for using \tilde{d}_j as a measure of the achievable gain with an improving vector \mathbf{a}_j . Recalling the change of basic variables as a function of the displacement of nonbasic variable x_j , which is $\boldsymbol{\beta}(t) = \boldsymbol{\beta} - t \boldsymbol{\alpha}_j$, the total distance the basic variables move is

$$D(t) = \|\boldsymbol{\beta}(t) - \boldsymbol{\beta}\| = t \|\boldsymbol{\alpha}_j\|,$$

assuming $t > 0$. If we consider the objective value as the function of this distance, i.e., $z = z(D(t))$, then its gradient at $t = 0$, using the chain rule, is

$$\begin{aligned}
 [\nabla z(D(t))]_{t=0} &= \left[\frac{dz(D(t))}{dD(t)} \right]_{t=0} \\
 &= \left[\frac{dz(D(t))}{dt} : \frac{dD(t)}{dt} \right]_{t=0} \\
 &= \frac{d_j}{\|\alpha_j\|} = \tilde{d}_j.
 \end{aligned} \tag{9.51}$$

The main conceptual difference between d_j and \tilde{d}_j is that the former indicates the change to the objective per unit change along the j -th coordinate axis, while the latter is the initial rate of change in the objective value with respect to the change in the total distance. In other words, d_j does not take into account the movement of other variables while \tilde{d}_j does. The normalized measure will certainly not favor columns which have large d_j but also large coefficients in α_j .

The above idea can be thought of as if we had applied a proper weight on d_j to obtain \tilde{d}_j . While \tilde{d}_j is a more promising indicator of the overall profitability of an improving candidate than d_j , there is a considerable difficulty in using it. Namely, it requires prohibitively much extra work per iteration if we try to compute it according to (9.50). It is equivalent to maintaining the full updated simplex tableau. There are several ways to alleviate this problem. Some methods have been developed to approximate $\|\alpha_j\|$ and there are also exact methods that obtain $\|\alpha_j\|$ by updating it rather than recomputing.

The approximate methods involve different amounts of extra work per iteration and they also require the periodical recalculation or resetting of the weights. Because of the importance of dynamic scaling we present several methods starting with an exact steepest edge technique.

9.5.4.1 Steepest edge

The steepest edge pricing for the primal simplex method was first described by Goldfarb and Reid in [Goldfarb and Reid, 1977]. Here, we present the main idea and the detailed computational development of the method.

Temporarily, it is assumed the model does not contain upper bounded variables. Thus, the problem is $\min\{\mathbf{c}^T \mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0\}$. For simplicity, the basis consists of the first m columns and \mathbf{A} is partitioned as $[\mathbf{B} \mid \mathbf{R}]$. The basic feasible solution at the start of the iteration is

$$\mathbf{x} = \begin{bmatrix} \mathbf{B}^{-1}\mathbf{b} \\ \mathbf{0} \end{bmatrix}.$$

The edge directions emanating from \mathbf{x} are the column vectors of the nonbasic variables in the extended simplex tableau, i.e.,

$$\boldsymbol{\sigma}_j = \begin{bmatrix} -\mathbf{B}^{-1}\mathbf{R} \\ \mathbf{I}_{n-m} \end{bmatrix} \mathbf{e}_{j-m} = \begin{bmatrix} -\boldsymbol{\alpha}_j \\ \mathbf{e}_{j-m} \end{bmatrix}, \quad j = m+1, \dots, n, \quad (9.52)$$

where \mathbf{e}_i is the i -th column of \mathbf{I}_{n-m} . Note, now $j \in \mathcal{R} \equiv m < j \leq n$. If edge $q \in \mathcal{R}$ is chosen with a steplength of θ then the next iterate is

$$\bar{\mathbf{x}} = \mathbf{x} + \theta \boldsymbol{\sigma}_q.$$

As an improving edge direction must be downhill (minimization problem), it must make an obtuse angle with \mathbf{c} , i.e., $d_q = \mathbf{c}^T \boldsymbol{\sigma}_q = c_q - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{a}_q < 0$ must hold. From among all downhill directions the steepest edge algorithm chooses one that is most negative relative to $\|\boldsymbol{\sigma}_q\|$:

$$\frac{\mathbf{c}^T \boldsymbol{\sigma}_q}{\|\boldsymbol{\sigma}_q\|} = \min_{j \in \mathcal{R}} \left\{ \frac{\mathbf{c}^T \boldsymbol{\sigma}_j}{\|\boldsymbol{\sigma}_j\|} \right\}.$$

The usefulness of this approach heavily depends on how quantities $\|\boldsymbol{\sigma}_j\|$ are computed. Note, (9.52) shows there is a close relationship between $\|\boldsymbol{\sigma}_j\|$ and $\|\boldsymbol{\alpha}_j\|$, namely $\|\boldsymbol{\sigma}_j\|^2 = \|\boldsymbol{\alpha}_j\|^2 + 1$.

Using notation $\gamma_j = \|\boldsymbol{\sigma}_j\|^2 = \boldsymbol{\sigma}_j^T \boldsymbol{\sigma}_j$ recurrence formulas can be developed for updating these quantities after every basis change. Denote the selected incoming variable by x_q and the leaving variable by x_p (which is now the p -th basic variable).

The new edge directions are determined using the transformation formulas of the simplex method:

$$\bar{\boldsymbol{\sigma}}_p = -\frac{1}{\alpha_q^p} \boldsymbol{\sigma}_q \quad (9.53)$$

$$\bar{\boldsymbol{\sigma}}_j = \boldsymbol{\sigma}_j - \frac{\alpha_j^p}{\alpha_q^p} \boldsymbol{\sigma}_q, \quad j \in \mathcal{R}, \quad j \neq q. \quad (9.54)$$

As the multiplier of $\boldsymbol{\sigma}_q$ in (9.54) is equal to the updated element in the pivot row after the basis change, it can be denoted by $\bar{\alpha}_j^p$. With this notation (9.54) can be rewritten as $\bar{\boldsymbol{\sigma}}_j = \boldsymbol{\sigma}_j - \bar{\alpha}_j^p \boldsymbol{\sigma}_q$. The updated γ -s can now be determined:

$$\bar{\gamma}_p = \frac{1}{(\alpha_q^p)^2} \gamma_q \quad (9.55)$$

$$\begin{aligned} \bar{\gamma}_j &= \bar{\boldsymbol{\sigma}}_j^T \bar{\boldsymbol{\sigma}}_j \\ &= (\boldsymbol{\sigma}_j - \bar{\alpha}_j^p \boldsymbol{\sigma}_q)^T (\boldsymbol{\sigma}_j - \bar{\alpha}_j^p \boldsymbol{\sigma}_q) \\ &= \boldsymbol{\sigma}_j^T \boldsymbol{\sigma}_j - 2\bar{\alpha}_j^p \boldsymbol{\sigma}_j^T \boldsymbol{\sigma}_q + (\bar{\alpha}_j^p)^2 \boldsymbol{\sigma}_q^T \boldsymbol{\sigma}_q \\ &= \gamma_j + (\bar{\alpha}_j^p)^2 \gamma_q - 2\bar{\alpha}_j^p \boldsymbol{\sigma}_j^T \boldsymbol{\sigma}_q, \quad j \in \mathcal{R}, \quad j \neq q. \end{aligned} \quad (9.56)$$

The unit coefficients of σ_j and σ_q in (9.56) are in different positions if $j \neq q$. Therefore, (9.52) implies $\sigma_j^T \sigma_q = \alpha_j^T \alpha_q$. To bring the latter into a computationally more advantageous form note that, from $\alpha_j = \mathbf{B}^{-1} \mathbf{a}_j$, we have $\alpha_j^T = \mathbf{a}_j^T (\mathbf{B}^{-1})^T = \mathbf{a}_j^T \mathbf{B}^{-T}$. Consequently, (9.56) can be replaced by

$$\bar{\gamma}_j = \gamma_j + (\bar{\alpha}_j^p)^2 \gamma_q - 2\bar{\alpha}_j^p \mathbf{a}_j^T \mathbf{B}^{-T} \alpha_q, \quad j \in \mathcal{R}, \quad j \neq q. \quad (9.57)$$

Now we are ready to assess the computational effort needed to use (9.55) and (9.57) for updating the norms. First of all, the pivot column, α_q , is available as it was needed for the ratio test. It enables us to compute $\gamma_q = \alpha_q^T \alpha_q + 1$ directly. The next concern is $\bar{\alpha}_j^p$ in (9.57). It is the element of the p -th row in $\bar{\mathbf{B}}^{-1} \mathbf{A}$. It can be obtained by multiplying row p of the inverse of the new basis $\bar{\mathbf{B}}^{-1}$ by column j . It can also be determined by remembering that $\bar{\alpha}_j^p = \alpha_j^p / \alpha_q^p$. In this case quantities α_j^p and α_q^p are defined in the old basis \mathbf{B} . Row p of \mathbf{B}^{-1} can be obtained by $\mathbf{v}^T = \mathbf{e}_p^T \mathbf{B}^{-1}$ which requires the BTRAN of a unit vector. If either the simplex multiplier or the reduced costs are updated (see section 9.4) then this computation is done anyhow. Therefore, it emerges as extra work only if none of them is used. The most critical operation in (9.57) is computing the $\mathbf{a}_j^T \mathbf{B}^{-T} \alpha_q$ term. The $\mathbf{B}^{-T} \alpha_q$ part of it can be obtained as

$$\mathbf{w}^T = \alpha_q^T \mathbf{B}^{-1}, \quad (9.58)$$

i.e., a BTRAN of a probably quite dense vector α_q . Then it is followed by a

$$\mathbf{w}^T \mathbf{a}_j \quad (9.59)$$

dot product with every column $j \in \mathcal{R}$, $j \neq q$. Operations (9.58) and (9.59) are specific to the exact update of the steepest edge method. The (9.59) dot products may not be as heavy as they appear at the first glance. Namely, looking at the $2\bar{\alpha}_j^p \mathbf{a}_j^T \mathbf{B}^{-T} \alpha_q$ term in (9.57), the $\mathbf{w}^T \mathbf{a}_j$ computations have to be performed only for those columns for which $\bar{\alpha}_j^p \neq 0$. Experience shows that the updated pivot row can be very sparse, therefore, $\bar{\alpha}_j^p = 0$ can hold for many j indices.

Summarizing the above development, the exact update of the steepest edge column weights requires the following extra arithmetic operations counted in flops (a double precision multiplication and an addition).

- 1 $\gamma_q = \alpha_q^T \alpha_q + 1$ needs $O(m\rho_q)$ flops, where ρ_q is the density of α_q . The density of α_q can reach 50% for some problems.
- 2 $\mathbf{w}^T = \alpha_q^T \mathbf{B}^{-1}$ requires no more than $O(n_\eta(n_\eta + m\rho_q))$ flops, where n_η is the number of elementary matrices in the product form of the

inverse. This is very likely the single most expensive operation needed for the updating.

- 3 The number of flops to compute $2\bar{\alpha}_j^p \mathbf{a}_j^T \mathbf{B}^{-T} \boldsymbol{\alpha}_q$ depends on two factors. First, the density of the updated pivot row $\boldsymbol{\alpha}^p = \mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{R}$ which accounts for the number of nonzeros of $\bar{\alpha}_j^p$. In general it is rather low. Second, the number of nonzeros in the intersecting columns corresponding to $\bar{\alpha}_j^p \neq 0$. Obviously, it cannot be more than the total number of nonzeros in \mathbf{R} and can be much smaller. Experience shows that this potentially heavy operation remains well below the work of computing $\mathbf{w}^T = \boldsymbol{\alpha}_q^T \mathbf{B}^{-1}$.

Some reports suggest that the overall loss in iteration speed of the steepest edge method is about 10–30% if compared to the full Dantzig pricing. It can be achieved by using carefully implemented software engineering techniques and, maybe, taking advantage of a given hardware platform. The gain in the number of iterations can be substantial.

In practice, to avoid inconvenience caused by numerical inaccuracies (like $\bar{\gamma}_j < 0$) the formula under (9.57) can be replaced by

$$\bar{\gamma}_j = \max\{\gamma_j + (\bar{\alpha}_j^p)^2 \gamma_q - 2\bar{\alpha}_j^p \mathbf{a}_j^T \mathbf{B}^{-T} \boldsymbol{\alpha}_q, (\bar{\alpha}_j^p)^2 + 1\}, \quad j \in \mathcal{R}, \quad j \neq q,$$

as $\bar{\gamma}_j \geq (\bar{\alpha}_j^p)^2 + 1$. See (9.73) for the development of this lower bound.

If there are upper bounded variables included in the problem then the above developments remain valid unchanged for the following reasons. Let $0 \leq x_j \leq u_j$. If x_j is nonbasic at lower bound then the downhill edge directions are the same as above. If $x_j = u_j$ was chosen then the edge direction is the negative of that of the former case. Therefore, as the size of the corresponding σ_j remains the same the quantities γ_j and the (9.55) and (9.57) update formulas do not change. The case of free variables can be argued similarly.

9.5.4.2 Devex

Historically the Devex method developed by Harris [Harris, 1973] was the first normalized pricing algorithm. In Devex the scale factors of the reduced costs are column weights determined in a reference coordinate system in the sense of (9.50) or (9.51). The weights are then approximately updated after each basis change. As approximations become increasingly inaccurate a resetting of the reference framework and the weights is arranged.

To describe Devex we use similar assumptions as we did at the steepest edge algorithm. Namely, $\mathbf{A} = [\mathbf{B} \mid \mathbf{R}]$ and the edge directions are also denoted by σ_j as in (9.52). The algorithm uses weights t_j to approximate

the norms of the subvectors, denoted by \mathbf{s}_j , consisting of only those components of $\boldsymbol{\sigma}_j$ that belong to the current reference framework. When a new reference framework is fixed it consists of the actual nonbasic variables and, consequently, the weights are set equal to one. At this point all weights are exact, which is obvious if we consider the extended simplex tableau, though in the first iteration after resetting they will not really scale the computed d_j values (being $t_j = 1$). The original updating of Devex can be derived in the following way.

Let \mathcal{H} denote the index set of variables in the reference framework. In the description we use t_j to denote the square of the weights

$$t_j = \mathbf{s}_j^T \mathbf{s}_j = \sum_{k_i \in \mathcal{H}} (\alpha_j^i)^2, \quad (9.60)$$

where k_i is the index of the variable that is basic in position i .

As the updated pivot column $\boldsymbol{\alpha}_q$ is available t_q can be determined exactly and compared to its approximation. Having performed a basis change, the coefficients of a nonbasic column are transformed according to the simplex rules:

$$\bar{\alpha}_j^p = \frac{\alpha_j^p}{\alpha_q^p} \quad (9.61)$$

$$\bar{\alpha}_j^i = \alpha_j^i - \bar{\alpha}_j^p \alpha_q^i \quad \text{for } i \neq p. \quad (9.62)$$

As the elements of the pivot row after the basis change are $\bar{\alpha}_j^p$, (9.61) and (9.62) together can be written in a single equation as

$$\bar{\boldsymbol{\alpha}}_j = \boldsymbol{\alpha}_j - \bar{\alpha}_j^p \boldsymbol{\alpha}_q + \bar{\alpha}_j^p \mathbf{e}_p \quad (9.63)$$

with \mathbf{e}_p being the p -th m dimensional unit vector. Extracting the coefficients of the reference framework, the updated weights are

$$\begin{aligned} \bar{t}_j &= \bar{\mathbf{s}}_j^T \bar{\mathbf{s}}_j \\ &= \mathbf{s}_j^T \mathbf{s}_j + (\bar{\alpha}_j^p)^2 \mathbf{s}_q^T \mathbf{s}_q + (\bar{\alpha}_j^p)^2 - 2\bar{\alpha}_j^p \mathbf{s}_j^T \mathbf{s}_q + 2\bar{\alpha}_j^p \alpha_j^p - 2(\bar{\alpha}_j^p)^2 \alpha_q^p \\ &= t_j + (\bar{\alpha}_j^p)^2 (t_q + 1) - 2\bar{\alpha}_j^p \mathbf{s}_j^T \mathbf{s}_q. \end{aligned} \quad (9.64)$$

At this stage Harris drops the last term in (9.64) and uses the approximation

$$\bar{t}_j \approx t_j + (\bar{\alpha}_j^p)^2 (t_q + 1) \quad \forall j \neq q. \quad (9.65)$$

As a further simplification (and further approximation), (9.65) is reduced to

$$\bar{t}_j = \max\{t_j, (\bar{\alpha}_j^p)^2 t_q\} \quad \forall j \neq q. \quad (9.66)$$

Finally, the weight of the outgoing variable is approximated by

$$\bar{t}_p = \max \left\{ 1, \frac{t_q}{(\alpha_q^p)^2} \right\}. \quad (9.67)$$

As a summary, the squares of the Devex weights are transformed after a basis change according to (9.66) and (9.67). To facilitate the computations the α_j^p elements of the updated pivot row have to be determined. If the row of reduced costs is kept in an updated form then these coefficients are available and the extra work is minimal. Otherwise, the $\alpha_j^p = \mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{a}_j = \mathbf{v}^T \mathbf{a}_j$ values have to be determined by first computing $\mathbf{v}^T = \mathbf{e}_p^T \mathbf{B}^{-1}$ (BTRAN operation) and then performing a dot product with every nonbasic column. In this case the amount of extra work is reduced if the simplex multiplier π is kept in an updated form as it requires the use of the same \mathbf{v} .

From (9.66) it follows that the Devex weights are monotonically increasing. On the other hand, the weight of the incoming column can be determined exactly, see (9.60) with $j = q$. This observation makes it possible to assess the accuracy of the approximation. If the estimated and the computed values differ by a significant margin a resetting of the reference framework and the weights is done as described above. A practical test for the accuracy is that if the ratio of the estimated to the computed weight is greater than 3 then a resetting is triggered. The critical ratio is usually a user accessible parameter.

Devex can be viewed as an approximation to the steepest edge. In its current form it can be very effective in reducing the number of iterations at the expense of the above extra computations. However, steepest edge makes nearly always fewer iterations at the expense of more extra computations per iteration. The tradeoff is problem specific. In a sophisticated implementation both methods must be present ready for action when needed.

To conclude the discussion of Devex, we show how the Devex weights can be updated in an exact way. This development is due to Greenberg and Kalan [Greenberg and Kalan, 1975]. They investigated the dropped term $2\bar{\alpha}_j^p \mathbf{s}_j^T \mathbf{s}_q$ in (9.64). To simplify notations, we use the full form of the norm and denote it by T_j to have

$$T_j = \sum_{i=1}^m (\alpha_j^i)^2.$$

The norms are updated according to (9.64) to give

$$\bar{T}_j = T_j + (\bar{\alpha}_j^p)^2 (T_q + 1) - 2\bar{\alpha}_j^p \boldsymbol{\alpha}_j^T \boldsymbol{\alpha}_q. \quad (9.68)$$

For the outgoing variable the α vector is equal to e_p as this variable was basic in position p . Therefore its T_j was equal to 1 which means

$$\bar{T}_j = 1 + (\bar{\alpha}_j^p)^2(T_q + 1) - 2\bar{\alpha}_j^p e_p^T \alpha_q$$

leaving the dropped term

$$2\bar{\alpha}_j^p e_p^T \alpha_q = 2 \frac{1}{\alpha_q^p} \alpha_q^p = 2.$$

This is quite a significant number. Therefore, one immediate improvement to Harris' scheme is to use of this exact value for the outgoing variable. Additionally, it is possible to determine the $\alpha_j^T \alpha_q$ term in (9.68) exactly for the other nonbasic variables. Expressing α_j from the (9.63) update formula and substituting it into $\alpha_q^T \alpha_j$:

$$\begin{aligned} \alpha_q^T \alpha_j &= \alpha_q^T (\bar{\alpha}_j + \bar{\alpha}_j^p \alpha_q - \bar{\alpha}_j^p e_p) \\ &= \alpha_q^T \bar{\alpha}_j + \bar{\alpha}_j^p T_q - \bar{\alpha}_j^p \alpha_q^p. \end{aligned} \quad (9.69)$$

In (9.69) every item is known except $\alpha_q^T \bar{\alpha}_j$. However, as $\alpha_q^T \bar{\alpha}_j = \alpha_q^T \mathbf{B}^{-1} \mathbf{a}_j$ it can also be determined at the expense of extra computations. First $\mathbf{w}^T = \alpha_q^T \mathbf{B}^{-1}$ needs to be determined (BTRAN of α_q) which is then followed by the dot products $\mathbf{w}^T \mathbf{a}_j$ for each nonbasic variable. This latter computation is specific to the exact update of the Devex weights but shows a very obvious similarity to the steepest edge update. Clearly, the Devex computations are performed only for those positions of the appropriate α vectors that belong to the current reference framework.

9.5.4.3 Modified Devex

Benichou et al. [Benichou et al., 1977] have experimented with a modified version of Devex. They define the column scale as

$$t_j = \max \left\{ 1, \sum_{k_i \in \mathcal{H}} |\alpha_j^i| \right\}, \quad (9.70)$$

where k_i is the subscript of the variable that is basic in position i and \mathcal{H} is a reference framework as in Devex. To approximate the sum in (9.70) an auxiliary vector $\mathbf{v} \in \mathbb{R}^m$ is introduced such that its i -th component is equal to one if the i -th basic variable is in the reference framework and zero otherwise. More formally,

$$v_i = \begin{cases} 1, & \text{if } k_i \in \mathcal{H}, \\ 0, & \text{otherwise.} \end{cases}$$

With this vector the following sum can be computed:

$$\sum_{k_i \in \mathcal{H}} \alpha_j^i = \mathbf{v}^T \mathbf{B}^{-1} \mathbf{a}_j = \mathbf{v}^T \boldsymbol{\alpha}_j.$$

The absolute value of this sum can be used as an approximation of $\sum_{k_i \in \mathcal{H}} |\alpha_j^i|$ in (9.70). Obviously, this is usually a rough approximation of the column norm. It has, however, some advantages.

This modified Devex method is very simple and the computational overhead is moderate. It can easily be incorporated in any partial and/or multiple pricing scheme.

The method requires the recomputation or updating of the $\mathbf{v}^T \mathbf{B}^{-1}$ vector in each (major) iteration. Additionally, dot products have to be performed with columns that have profitable d_j -s.

In practice, the performance of the modified Devex is inferior to other normalized pricing methods. However, Maros and Mitra [Maros and Mitra, 2000] have shown that there can be cases when its advantages outweigh its weaknesses. Therefore, it is worth including it in an advanced implementation of the simplex method so that it can be activated if it appears beneficial.

9.5.4.4 Another steepest edge approximation

The most critical term in the exact update of the column norms in the steepest edge method was the computation of the $\bar{\alpha}_j^p \boldsymbol{\sigma}_j^T \boldsymbol{\sigma}_q$ term in (9.56) or, equivalently, $\bar{\alpha}_j^p \mathbf{a}_j^T \mathbf{B}^{-T} \boldsymbol{\alpha}_q$ in (9.57). Swietanowski has proposed an approximation scheme for it in [Swietanowski, 1998] which he calls *ASE*. It works in the following way.

The $\boldsymbol{\sigma}_j^T \boldsymbol{\sigma}_q$ term for $j \neq q$ is split up into two:

$$\boldsymbol{\sigma}_j^T \boldsymbol{\sigma}_q = \boldsymbol{\alpha}_j^T \boldsymbol{\alpha}_q = \alpha_j^p \alpha_q^p + \sum_{\substack{i=1 \\ i \neq p}}^m \alpha_j^i \alpha_q^i \quad (9.71)$$

and the

$$\boldsymbol{\alpha}_j^T \boldsymbol{\alpha}_q \approx \alpha_j^p \alpha_q^p = \bar{\alpha}_j^p (\alpha_q^p)^2 \quad (9.72)$$

approximation is used. The assumption behind this approximation is that $\boldsymbol{\alpha}_j$ may be very sparse and most of the $\alpha_j^i \alpha_q^i$ products are zero. A lower bound on the exact weight γ_j is easily derived. As $\gamma_j = \boldsymbol{\sigma}_j^T \boldsymbol{\sigma}_j = \boldsymbol{\alpha}_j^T \boldsymbol{\alpha}_j + 1$, using (9.71) and (9.72), we obtain

$$\gamma_j \geq (\alpha_j^p)^2 + 1 = (\bar{\alpha}_j^p)^2 (\alpha_q^p)^2 + 1. \quad (9.73)$$

Denoting the approximate weights by g_j , the following update formula can be used to replace (9.57) for $j \neq q$:

$$\bar{g}_j = \max\{g_j, (\bar{\alpha}_j^p)^2(\alpha_q^p)^2 + 1\} - 2(\bar{\alpha}_j^p)^2(\alpha_q^p)^2 + (\bar{\alpha}_j^p)^2\gamma_q, \quad (9.74)$$

as γ_q can be computed as in the exact steepest edge method. For $j = q$ the exact formula works in a straightforward way:

$$\bar{g}_q = \frac{\gamma_q}{(\alpha_q^p)^2}.$$

(9.74) enables us to give a lower bound on the approximate weights \bar{g}_j :

$$\begin{aligned} \bar{g}_j &\geq (\bar{\alpha}_j^p)^2(\alpha_q^p)^2 + 1 - 2(\bar{\alpha}_j^p)^2(\alpha_q^p)^2 + (\bar{\alpha}_j^p)^2\gamma_q \\ &\geq (\bar{\alpha}_j^p)^2(\alpha_q^p)^2 + 1 - 2(\bar{\alpha}_j^p)^2(\alpha_q^p)^2 + (\bar{\alpha}_j^p)^2((\alpha_q^p)^2 + 1) \\ &= (\bar{\alpha}_j^p)^2 + 1. \end{aligned} \quad (9.75)$$

It means the g_j weights are never smaller than one.

Comparing this approximation with Devex, it is to be noted that while the approximate weights in Devex are monotonically increasing, the g_j weights can increase or decrease (as it is obvious from (9.74)) but they never go below one, as just pointed out in (9.75).

As this method is also an approximation procedure for updating the steepest edge weights the accuracy may be tested in a similar way as in Devex. It is also possible to check the accuracy of the reduced cost of the incoming variable (comparing its updated value against its recomputed value when the incoming vector is updated by FTRAN). If it is found unsatisfactory it can be assumed that the update of the weights has also become inaccurate, a cause for resetting.

It can also be said that we are not interested in the accurate values of the weights but rather in the relative increase or decrease of possible profitability of introducing a nonbasic variable into the basis. Assume the problem was scaled terminating with equilibration, thus $\max_{i,j} |a_{ij}^i| = 1$. For this case the approximate weights can be set to the number of nonzeros in each column plus one. This setting will favor the sparse nonbasic variables to enter the basis in the forthcoming iterations which can lead to faster iteration speed, at least for a while.

The extra computational work can be assessed from (9.74). It is not more than in case of Devex as only the updated elements of the pivot row are needed. These elements are computed anyhow if the reduced costs are updated in every iteration.

9.5.5 A general pricing framework

In this section we present a pricing scheme that creates a great flexibility and contains many known pricing strategies as special cases. It

also makes it possible to define new pricing strategies. The method has been proposed by Maros [Maros, 1990] and [Maros, 2002a].

In most large scale LP problems the variables can be grouped into disjoint clusters according to the distribution of the nonzeros in their column vectors. The main characteristics of variables in a cluster is that they are somehow related to each other. On the other hand, variables in different cluster are relatively unrelated in the sense that they do not consume the same b_i resources (their nonzeros are in different rows). Typical examples are problems that represent dynamic models, spatial distribution or uncertainty.

To describe the framework the following notations are used

- \mathcal{N} : the index set of all variables in the problem,
- K : the number of disjoint clusters of variables,
- \mathcal{N}_k , $k = 1, \dots, K$: the index set of variables in cluster k ,
- $n_k = |\mathcal{N}_k| > 0$: the cardinality of \mathcal{N}_k .

The set of indices of all variables is thus

$$\mathcal{N} = \bigcup_{k=1}^K \mathcal{N}_k, \quad \mathcal{N}_i \cap \mathcal{N}_j = \emptyset, \quad \text{if } i \neq j.$$

Obviously, $|\mathcal{N}| = n$.

We consider a fixed circular ordering of the variables within each cluster, and also a fixed circular ordering of the clusters. It can be assumed that the members of a cluster are located contiguously. If not, a linked list structure can be created and the variables can be accessed by some fixed order in this list.

Common sense suggests and experience shows that if a properly selected vector from a cluster enters the basis at a given iteration then it is unlikely that good candidates in the same cluster (or in clusters that have also recently been visited) can be found during the next pricing pass. This can be interpreted in such a way that if the—locally—best vector enters the basis then, as a consequence of the correlation among the members of the cluster, the other candidates usually lose their attractiveness as they have become represented in the current solution by their ‘best’ candidate. If so, we can avoid superfluous pricing of non-improving or little-improving vectors by starting to look for new candidates in a different cluster(s). At the same time, we do not have to visit all the clusters if a sufficient number of profitable candidates have been found (partial pricing of the clusters). Furthermore, it is not necessary to make a complete scan of a cluster either. We can stop scanning

it if an acceptable number of candidates have been found there (partial pricing of a cluster).

On the basis of what we have said so far, we can introduce the following pricing (column selection) algorithm, referenced as SIMPRI (for SIMplex PRIcing) in the sequel.

The main parameters that control SIMPRI are:

- K : the number of clusters (defined above),
- S : the number of clusters to be scanned in one iteration ($1 \leq S \leq K$),
- V : the number of improving vectors to be found in one cluster (this number can be V_k , if we allow it to be different for each cluster).

The i -th element of cluster k in the fixed circular traversing order will be denoted by e_i^k . Two pointers are also needed: k will point to a cluster, while i_k will point to an element in cluster k .

In a general step of the algorithm pricing stopped in \mathcal{N}_k at $e_{i_k}^k$. The next pricing will start in the next cluster \mathcal{N}_{k+1} at the next element $e_{i_{k+1}+1}^{k+1}$ to come after the one where pricing stopped at the last visit in cluster \mathcal{N}_{k+1} . It means that if we increment k by 1 ($\text{mod } K$) and i_k by 1 ($\text{mod } n_k$) we can start pricing formally in \mathcal{N}_k at $e_{i_k+1}^k$.

For the description of the complete algorithm in which this general step is imbedded, some auxiliary variables have to be introduced:

- v is the number of improving variables found so far within a cluster during the current pass of SIMPRI,
- s counts the number of clusters scanned,
- t counts the number of vectors interrogated in a cluster
- d_{\max} is the largest reduced cost found so far,
- j is the absolute index ($1 \leq j \leq n$) of d_{\max} .

Algorithm SIMPRI:

Step 0. Setup. Initialize

pointers: $k = K$, $i_k = n_k$, $k = 1, \dots, K$,

counts and markers: $s = 0$, $d_{\max} = 0$, $j = 0$.

Step 1. Cluster count. Increment s . If $s > K$ go to Step 7.

Step 2. Cluster initialization. Increment k ($\text{mod } K$). Set $v = 0$, $t = 0$.

Step 3. Variable count. Increment t . If $t > n_k$ go to Step 6.

Step 4. *Reduced cost.* Increment $i_k \pmod{n_k}$.

If $e_{i_k}^k$ is nonbasic, compute its reduced cost d , otherwise go to Step 3.

If d is non-profitable then go to Step 3.

Increment v by 1.

If d is better than d_{\max} then

d is the new value of d_{\max} and the absolute index of $e_{i_k}^k$ is recorded in j .

Step 5. *Terminate cluster?* If the sufficient number (V) of improving vectors have been found exit from the cluster, i.e., if $v = V$ go to Step 6, otherwise go to Step 3.

Step 6. *Terminate pricing?* If the prescribed number (S) of clusters have been scanned and at least one candidate was found, terminate pricing, otherwise take next cluster, i.e., if $s \geq S$ and $d_{\max} > 0$ then go to Step 7, otherwise go to Step 1.

Step 7. *Evaluate termination.* If $d_{\max} \neq 0$ then variable with index j is a candidate to enter the basis, otherwise the simplex algorithm terminates. In the latter case if the current solution is feasible then it is optimal, if infeasible then the problem has no feasible solution.

Some comments on SIMPRI can help identify its general usefulness.

- If d_j is explicitly maintained then ‘compute reduced cost’ can be replaced by ‘retrieve reduced cost’.
- If normalized pricing is used it is usually done in the framework of full pricing. However, even in this case it is still possible to use some sort of a partial pricing. In either case, SIMPRI can be used with the evaluation criterion of the given normalized pricing method.
- SIMPRI is applicable to multiple pricing in a straightforward way. In this case, a pool of H best candidates is maintained, where H is a strategic parameter as discussed on page 186 in the introduction of section 9.5. Now the interesting issue is how to make the selection. It appears to be a good idea to keep candidates from different clusters with the understanding that they are relatively unrelated so that if one of them enters the basis the profitability of the other selected candidates will not be affected significantly.
- Pricing the nonbasic logical variables is computationally very simple and can easily be done whether explicit d_j -s are maintained or not. Logicals can be considered as a single cluster or, if there is information about a rowwise subdivision of \mathbf{A} , they can be added to the appropriate clusters.

It remains to see if SIMPRI really does not overlook any special situation that can occur. The answer is formulated in the following proposition.

PROPOSITION 9.5.1 *If the parameters are given meaningful values, i.e., $1 \leq K \leq n$, $1 \leq S \leq K$, and $1 \leq V \leq n$, then the application of SIMPRI in the simplex method leads to a correct answer to the LP problem.*

PROOF. It is enough to show that as long as there is at least one profitable candidate among the nonbasic variables, SIMPRI will find it, and SIMPRI will never fall into an endless loop.

First we show that if there is a single candidate it cannot be missed.

The algorithm will always start, because $K \geq 1$, therefore in the beginning relation “ $s > K$ ” in Step 1 is false, so Step 2 is performed.

Since it is assumed that $V \geq 1$, there is always something to look for. A candidate in a cluster cannot be overlooked: Variables in a cluster are scanned sequentially in a circular order by the inner loop through Steps 3 – 4 – 5. This loop can terminate in two ways: (i) through Step 5, if a sufficient number of profitable candidates have been found, (ii) through Step 3, if the list is exhausted, i.e., all the elements in the given cluster have been checked.

If no candidate was found in the requested (S) number of clusters then Step 6 and Step 1 ensure that new cluster(s), taken in the circular order, will be scanned as long as there are unvisited clusters.

What would make SIMPRI try to loop infinitely? It may be thought that such case could happen if SIMPRI is given an impossible task, like to find more profitable candidates than presently available. A typical situation of this kind occurs when we are at an optimal solution (i.e., there is no profitable candidate at all). An endless loop cannot occur within a cluster, Step 3 takes care of it, since in a cluster the basic variables also increment the pointer. Similarly, clusters cannot be taken infinitely one after the other since Step 1 does not allow more than K clusters to be scanned, regardless of the index of the starting cluster. These two remarks ensure the proper termination of SIMPRI. \square

It is clear that SIMPRI creates a large flexibility in pricing. It enables us to tune the simplex algorithm for identified problem families by proper setting of the newly introduced parameters.

Of course, it may happen that a structure does not exist at all. Even in this case, if $n \gg m$ holds, some kind of a partial pricing with SIMPRI can help a lot in reducing the overall effort to solve the problem. Maros and Mitra have observed [Maros and Mitra, 2000] that SIMPRI with reasonably chosen clusters (not necessarily closely following any ‘natural’ clustering) can, in fact, be very efficient.

It is instructive to see how some known pricing schemes can be obtained as special cases of SIMPRI by appropriate setting of the three parameters.

- 1 If we define \mathcal{N} as one cluster, requiring all nonbasic vectors to be scanned then we arrive at the generally known *full pricing*. This is achieved by the following setting of the parameters: $K = 1$, $S = 1$, and $V = n$. Clearly, here we do not expect that the number of improving vectors will be equal to n , but by this setting we can force the algorithm to scan all the nonbasic variables in every step. If the selection criterion is the unscaled max d_j then it reproduces the Dantzig pricing [Dantzig, 1963], see also section 9.5.1 of this book.
- 2 If we want to price one cluster fully per each iteration, taking one cluster after the other then we get the full *sectional pricing*. SIMPRI will work in this way for $K > 1$ if we set $S = 1$, and $V = n$.
- 3 Pricing required by Bland's rule can be obtained if we set $K = 1$, $S = 1$, $V = 1$ and restart SIMPRI in every iteration at Step 0.
- 4 If we have $K > 1$ and set $S = K$, $V = 1$, then we get the *one candidate from each subset* pricing strategy.
- 5 The popular simple *dynamic cyclic pricing* [Benichou et al., 1977] is obtained if we set $K = 1$, $S = 1$, and V to some value $1 \leq V < |\mathcal{S}|$, where $\mathcal{S} = \{j \mid j \text{ nonbasic}, d_j > 0 \text{ at a given step of the algorithm}\}$, i.e, the set of profitable candidates in the entire problem. \mathcal{S} is usually determined after every reinversion of the basis (see description of partial pricing in section 9.5.2).
- 6 Another example is Cunningham's LRC (Least Recently Considered) rule [Cunningham, 1979] for the network simplex method which is achieved by setting $K = 1$, $S = 1$, and $V = 1$.

On the basis of these examples we can say that SIMPRI is certainly a generalization of several well known pricing schemes. By other settings one can actually create new pricing strategies.

9.6. Improving an infeasible basic solution

The simplex method assumes the availability of a basic feasible solution. It is well known that the simplex method itself can be used to find one. This algorithmic part is called the *phase-1* of the simplex method. From this notion it is clear why iterations in the feasible domain were called phase-2. Introductory textbooks provide several phase-1 procedures. They can be used to determine a first feasible solution if one

exists or conclude that the problem is infeasible. However, they are usually impractical and inconvenient in situations that can occur during the computer solution of large scale LP problems. Additionally, their efficiency is questionable in many cases. This section presents a generalized phase-1 procedure which overcomes these problems and has several additional favorable features that are computationally very advantageous. The material in this section is based on Maros's paper [Maros, 1986]. Additional readings are [Wolfe, 1965, Greenberg, 1978b, Maros, 1981].

The rest of the discussion will refer to CF-1. Also, notation introduced in section 9.1 will be used throughout. A basis \mathcal{B} together with \mathcal{U} is called infeasible if there is at least one basic variable outside its feasibility range. More formally, a basis is infeasible if set $\mathcal{M} \cup \mathcal{P} \neq \emptyset$, where \mathcal{M} and \mathcal{P} are the index sets of the infeasible basic positions as defined in (9.9) and (9.10), respectively. Using notation $\beta = \mathbf{x}_{\mathcal{B}}$ we can simplify the reference to basic variables. As a measure of infeasibility we use the negative of the sum of violations:

$$w = \sum_{i \in \mathcal{M}} \beta_i - \sum_{i \in \mathcal{P}} (\beta_i - v_i). \quad (9.76)$$

Obviously, $w \leq 0$. If $w = 0$ then both \mathcal{M} and \mathcal{P} are empty and the solution is feasible. Therefore, we can formulate a phase-1 problem

$$\max \quad w \quad (9.77)$$

$$\text{s.t.} \quad \mathbf{Ax} = \mathbf{b}, \quad (9.78)$$

$$\ell \leq \mathbf{x} \leq \mathbf{u}, \quad (9.79)$$

where some components of ℓ and \mathbf{u} can be $-\infty$ or $+\infty$, respectively. w is called the *phase-1 objective function*. This is not a conventional LP problem because the composition of the objective function can change in every iteration as sets \mathcal{M} and \mathcal{P} change.

Traditional phase-1 methods that allow infeasible basic variables work under the following conditions:

- 1 The incoming variable enters the basis at a feasible level.
- 2 The already feasible basic variables remain feasible (the number of infeasibilities does not increase).
- 3 The outgoing variable leaves the basis at a feasible level (lower or upper bound).

These conditions still do not determine the outgoing variable uniquely for a given incoming variable. Further possibilities arise if condition 2 is relaxed.

We can investigate how feasibility of the basis is affected if a nonbasic variable is displaced by t in the feasible direction. First, assume the displacement is nonnegative, $t \geq 0$. If x_q changes to $x_q + t$ the basic variables also change to maintain equations in (9.8). A development, identical to (9.17) – (9.19), gives the i -th basic variable as a function of t

$$\beta_i(t) = \beta_i - t\alpha_q^i. \quad (9.80)$$

The nonbasic variables that can improve the sum of infeasibilities can be identified on the basis of the following lemma.

LEMMA 9.1 *Given $\mathcal{M} \cup \mathcal{P} \neq \emptyset$ and $j \in \mathcal{R}$ with $x_j = 0$. w can be improved by increasing the value of x_j only if*

$$d_j = \sum_{i \in \mathcal{M}} \alpha_j^i - \sum_{i \in \mathcal{P}} \alpha_j^i < 0. \quad (9.81)$$

PROOF. Assume $t > 0$ is small enough so that sets \mathcal{M} and \mathcal{P} remain unchanged. In this case, taking (9.80) into account, the change of w is given by

$$\begin{aligned} \Delta w &= \sum_{i \in \mathcal{M}} [\beta_i(t) - \beta_i(0)] - \sum_{i \in \mathcal{P}} \{[\beta_i(t) - v_i] - [\beta_i(0) - v_i]\} \\ &= \sum_{i \in \mathcal{M}} (-t\alpha_j^i) - \sum_{i \in \mathcal{P}} (-t\alpha_j^i) \\ &= -t \left(\sum_{i \in \mathcal{M}} \alpha_j^i - \sum_{i \in \mathcal{P}} \alpha_j^i \right) \\ &= -td_j. \end{aligned} \quad (9.82)$$

Hence $d_j < 0$ is necessary for $\Delta w > 0$. □

If the infeasibility sets \mathcal{M} and \mathcal{P} remain unchanged only for $t = 0$ then the basis is degenerate. In this case, if condition 2 is in force, x_j can enter the basis without displacement (i.e., at its current nonbasic level). However, relaxing this condition changes this limitation, as will be seen later.

A feasible displacement of a variable can also be negative. In this case, w can be improved by decreasing x_j if $d_j > 0$. This observation is a direct consequence of lemma 9.1.

The expression in (9.81) is denoted by d_j and it symbolizes the *phase-1 reduced cost* of variable x_j . It is, in fact, the one sided partial derivative of w with respect to x_j .

Depending on the type and status of nonbasic variables (therefore, the possible feasible displacement t), from the viewpoint of their capability to improve w they fall into one of the following categories:

Type(x_j)	Value	d_j	Improving displacement	Remark
0	$x_j = 0$	Any	0	Never enters
1	$x_j = 0$	< 0	+	
1	$x_j = u_j$	> 0	-	$j \in \mathcal{U}$
2	$x_j = 0$	< 0	+	
3	$x_j = 0$	$\neq 0$	+/-	

Obviously, because of condition 1, type-0 variables cannot contribute to the improvement of the phase-1 objective function. Therefore, they are not considered to enter the basis.

Phase-1 terminates when no improving candidates can be found. If, at this point, $w = 0$ (because sets \mathcal{M} and \mathcal{P} are empty) then the solution is feasible and phase-2 can commence. However, if $w < 0$ then there is no feasible solution because infeasibility cannot be improved (the problem is infeasible).

Note, in phase-1 unboundedness (theoretically) cannot occur as the objective function is bounded from above, namely $w \leq 0$ holds for any basis.

Step one of an iteration in phase-1 is checking the d_j values of the nonbasic variables. To obtain these values the following procedure can be used. Define vector $\mathbf{h} \in \mathbb{R}^m$ with components

$$h_i = \begin{cases} 1, & \text{if } i \in \mathcal{M}, \\ -1, & \text{if } i \in \mathcal{P}, \\ 0, & \text{otherwise.} \end{cases}$$

It is easy to see that

$$d_j = \mathbf{h}^T \boldsymbol{\alpha}_j = \mathbf{h}^T \mathbf{B}^{-1} \mathbf{a}_j.$$

In this expression $\mathbf{h}^T \mathbf{B}^{-1}$ is independent of j and will be denoted by ϕ^T . It can be computed at the beginning of an iteration, just like the phase-2 simplex multiplier π , by

$$\phi^T = \mathbf{h}^T \mathbf{B}^{-1}. \quad (9.83)$$

This ϕ is called *phase-1 simplex multiplier*. The phase-1 reduced costs can now be determined as $d_j = \phi^T \mathbf{a}_j$ for all nonbasic variables. This

is practically the same as pricing in phase-2 but with a different multiplier vector. Therefore, the selection of the incoming candidate(s) can be done with the same pricing algorithms as discussed in section 9.5. Exceptions are some normalized pricing methods (steepest edge, Devex, approximate steepest edge, but not ‘modified Devex’) where updating the weights is more complicated. For the same reason phase-1 reduced costs are also not updated and, therefore, not kept in explicit form but computed afresh from $d_j = \phi^T \mathbf{a}_j$.

Having settled the issue of identifying the profitable incoming candidates for improving the phase-1 objective function, we can now turn to determining the outgoing variable (if any). It is assumed that x_q is the selected incoming variable. For notational simplicity, in the rest of this and the entire next section, column index q is omitted from α_q^i and α_i is used instead.

Define the ‘ $-$ ’ and ‘ $+$ ’ operators to denote

$$K^- = \begin{cases} 0, & \text{if } K \geq 0, \\ K, & \text{if } K < 0 \end{cases} \quad (9.84)$$

and

$$K^+ = \begin{cases} K, & \text{if } K > 0, \\ 0, & \text{if } K \leq 0. \end{cases} \quad (9.85)$$

The measure of infeasibility as a function of t can be expressed as

$$\begin{aligned} w(t) &= \sum_{i \in \mathcal{I}_\ell} [\beta_i(t)]^- - \sum_{i \in \mathcal{I}_u} [\beta_i(t) - v_i]^+ \\ &= \sum_{i \in \mathcal{I}_\ell} [\beta_i - t\alpha_i]^- - \sum_{i \in \mathcal{I}_u} [\beta_i - t\alpha_i - v_i]^+, \end{aligned} \quad (9.86)$$

where $\mathcal{I}_\ell = \mathcal{I}_0 \cup \mathcal{I}_1 \cup \mathcal{I}_2$ (index set of basic variables with 0 lower bound) and $\mathcal{I}_u = \mathcal{I}_0 \cup \mathcal{I}_1$ (basic variables with finite upper bound).

As K^- and $-K^+$ are piecewise linear concave functions their sum also has this property. Applied to (9.86), it follows that $w(t)$ is a continuous piecewise linear concave function. It has break points at t values where the feasibility status of at least one of the basic variables changes.

First we investigate the case when $t \geq 0$.

Recall the change of basic variables as the function of the displacement of the incoming variable is expressed as $\beta_i(t) = \beta_i - t\alpha_i$. The feasibility status of $\beta_i(t)$ changes if it reaches or goes beyond its bound. It happens if either $\beta_i - t\alpha_i = 0$ or $\beta_i - t\alpha_i = v_i$. Therefore, the break points are defined as follows.

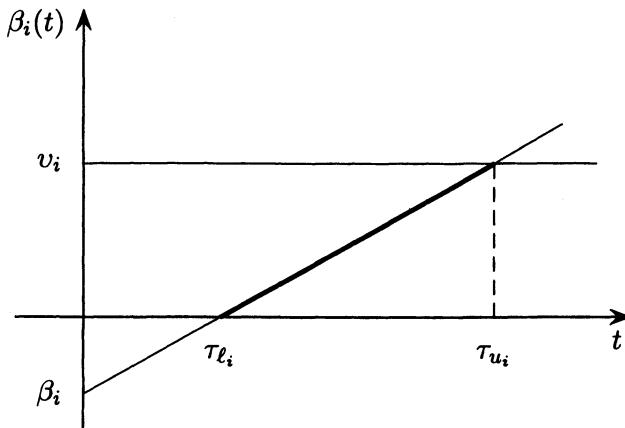


Figure 9.1. Definition of break points when $\alpha_i < 0$

Basic variables reach their lower bound or go below it at

$$\tau_{l_i} = \begin{cases} \beta_i/\alpha_i > 0 & \text{if } \alpha_i \neq 0 \text{ and } i \in \mathcal{I}_\ell, \\ 0 & \text{if } \beta_i = 0, \alpha_i > 0 \text{ and } i \in \mathcal{I}_\ell. \end{cases} \quad (9.87)$$

Basic variables reach their upper bound or go beyond it at

$$\tau_{u_i} = \begin{cases} (\beta_i - v_i)/\alpha_i > 0 & \text{if } \alpha_i \neq 0 \text{ and } i \in \mathcal{I}_u, \\ 0 & \text{if } \beta_i = 0, \alpha_i < 0 \text{ and } i \in \mathcal{I}_u. \end{cases} \quad (9.88)$$

If $v_i = 0$ then $\tau_{l_i} = \tau_{u_i}$ provided $\alpha_i \neq 0$. Among the ratios defined in (9.87) and (9.88) there can be several equal values. The multiplicity of a break point is the number of equal ratios defining it. Figures 9.1 and 9.2 give examples of how break points are defined. Note, in both examples the i -th basic variable is infeasible at $t = 0$. Figure 9.3 illustrates a situation where $\beta_i = \beta_i(0)$ is feasible. Several other cases exist that are not demonstrated here and they can easily be reconstructed based on these examples.

It is to be noted that type-0 and type-1 basic variables can define two break points (see figures 9.1 and 9.2), while type-2 variables at most one. Additionally, a type-0 variable always defines a break point if its α_q^i is nonzero. The maximum possible number of break points is $2m$ which is attained if all basic variables define two break points.

9.6.1 Analysis of $w(t)$

Relaxing condition 2 means a feasible basic variable may become infeasible. Of course, it makes sense only if this negative effect is compen-

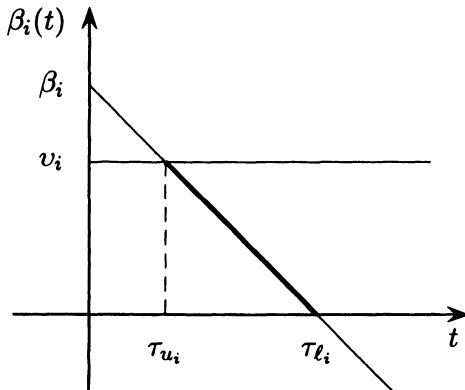


Figure 9.2. Definition of break points when $\alpha_i > 0$

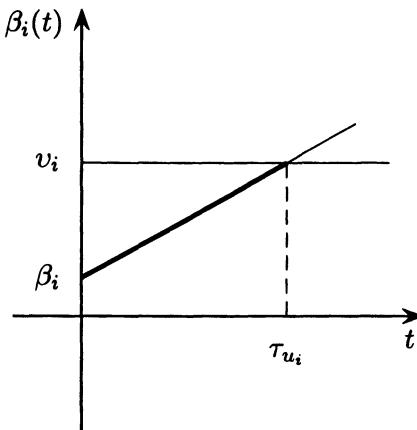


Figure 9.3. Definition of a break point when $\alpha_i < 0$ and β_i is feasible

sated, or rather outweighed, by some favorable ones. We set the main goal to find a displacement of the incoming variable which maximizes $w(t)$. As $w(t)$ is piecewise linear and concave function, if it has an extreme point it is a global maximum. It will be shown that the result of the maximization leads to a simplex-type iterative step. The procedure has some additional desirable properties that make it computationally very attractive.

From $\beta_i(t) = \beta_i - t\alpha_i$ it is obvious that the feasibility of the i -th basic variable as a function of t depends on its type, its ‘starting’ value (β_i), and the sign of α_i . If $\alpha_i > 0$ then $\beta_i(t)$ is a decreasing function while it

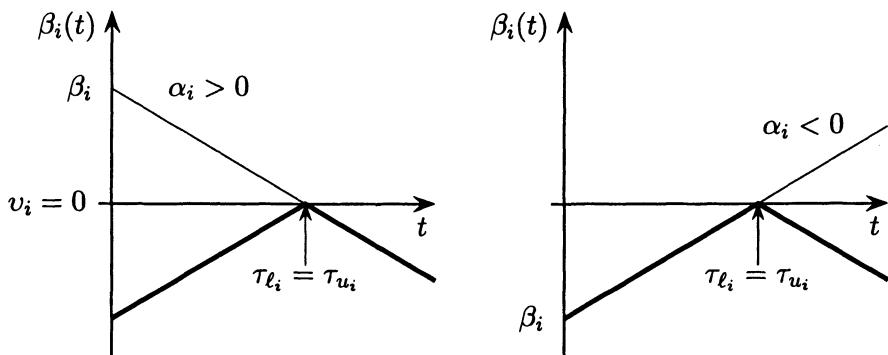


Figure 9.4. Thick line shows the contribution of a type-0 basic variable $\beta_i(t)$, $i \in \mathcal{I}_0$, to $w(t)$ if $\alpha_i > 0$ and $\alpha_i < 0$, respectively

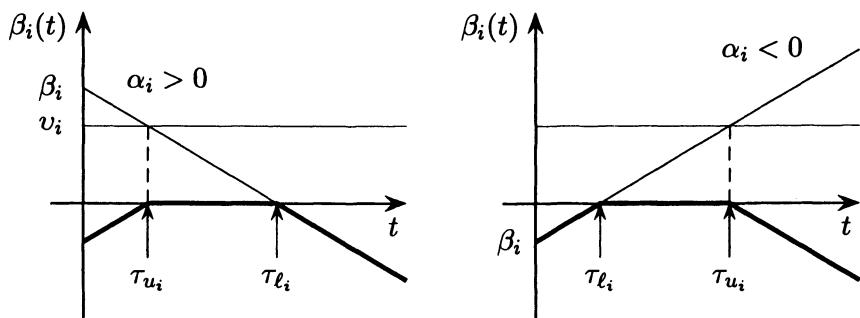


Figure 9.5. Thick line shows the contribution of a type-1 basic variable $\beta_i(t)$, $i \in \mathcal{I}_1$, to $w(t)$ if $\alpha_i > 0$ and $\alpha_i < 0$, respectively

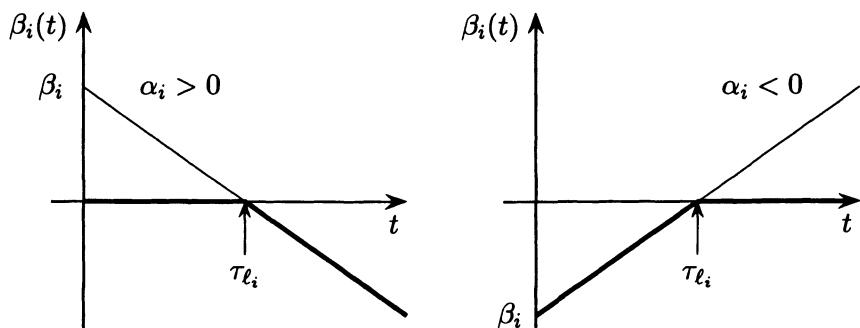


Figure 9.6. Thick line shows the contribution of a type-2 basic variable $\beta_i(t)$, $i \in \mathcal{I}_2$, to $w(t)$ if $\alpha_i > 0$ and $\alpha_i < 0$, respectively

is increasing if $\alpha_i < 0$. The contribution of the $\beta_i(t)$ to $w(t)$ is illustrated in figures 9.4, 9.5 and 9.6. Cases not shown (like the ones when β_i is feasible) can be derived as special cases of one of the represented ones.

When t moves away from 0 in the positive direction the first change in the feasibility status of one of the basic variables occurs when t reaches the first break point. It is the smallest of the ratios defined in (9.87) and (9.88). Since we want to pass this point and want to do further steps we assume that the break points are sorted into ascending order:

$$0 \leq t_1 \leq \dots \leq t_S,$$

where S denotes the number of break points and t is used with simple subscript to denote the elements of the ordered set of the τ_{ℓ_i} and τ_{u_i} values.

From lemma 9.1 we know that if x_q is the selected improving variable coming in from lower bound then the rate of change of $w(t)$ (the slope of $w(t)$) is $-d_q$. Therefore, in the $[0, t_1]$ interval $w(t)$ increases by $-d_q t_1$. Denoting

$$r_1 = -d_q = - \left(\sum_{i \in \mathcal{M}} \alpha_i - \sum_{i \in \mathcal{P}} \alpha_i \right) \quad (9.89)$$

the change is $r_1 t_1$ (remember the notation, α_i stands for α_q^i). At $t = t_1$ the feasibility status of at least one basic variable changes. Let the basic index of the variable that defines t_1 be denoted by j_1 . If $\alpha_{j_1} < 0$ then $\beta_{j_1}(t)$ reaches one of its bounds from below. Therefore, index j_1 moves either from \mathcal{M} to \mathcal{F} or (if $j_1 \in \mathcal{I}_u \cap \mathcal{F}$) from \mathcal{F} to \mathcal{P} . In both cases, from t_1 onwards, the slope of $w(t)$ will be $r_2 = r_1 + \alpha_{j_1}$ as can be seen from (9.89). Since α_{j_1} is negative, it means the slope of $w(t)$ decreases by $|\alpha_{j_1}|$. that is $r_2 = r_1 - |\alpha_{j_1}|$. If $\alpha_{j_1} > 0$ then $\beta_{j_1}(t)$ reaches one of its bounds from above. Thus, index j_1 moves either from \mathcal{P} to \mathcal{F} or from \mathcal{F} to \mathcal{M} . In both cases (see (9.89)) the slope of $w(t)$ will be $r_2 = r_1 - \alpha_{j_1}$ which can be rewritten as $r_2 = r_1 - |\alpha_{j_1}|$. Thus, we have obtained that in both cases the slope of $w(t)$ decreases by $|\alpha_{j_1}|$. Obviously, the same arguments are valid for any further t_k , $k = 2, \dots, S$. Therefore, the slope of $w(t)$ decreases at every break point t_k by $|\alpha_{j_k}|$ resulting in

$$r_{k+1} = r_k - |\alpha_{j_k}|, \quad k = 1, \dots, S$$

In the case of coinciding points the length of an interval may be zero.

Some characteristic shapes of $w(t)$ can be seen in figures 9.7, 9.8 and 9.9. Figure 9.7 shows a typical curve. Three linear sections are traversed and the maximum is reached at t_3 . In Figure 9.8 $w(t)$ reaches its theoretical maximum of 0. Figure 9.9 shows an interesting phenomenon. Here we find $t_1 = 0$, meaning the basis is degenerate as

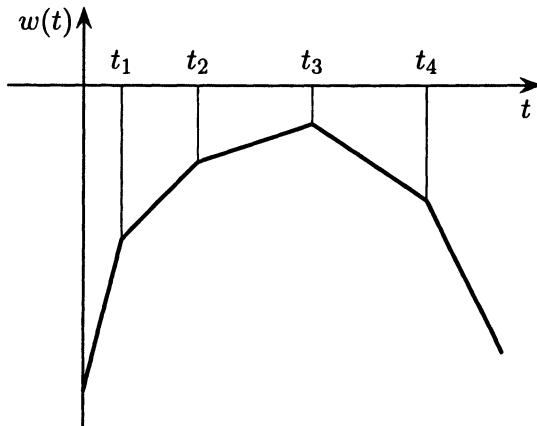


Figure 9.7. Typical shape of $w(t)$.

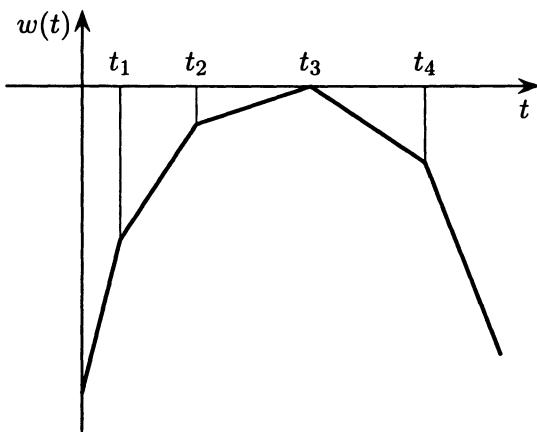


Figure 9.8. $w(t)$ reaches its theoretical maximum of 0 at t_3 .

either $t_1 = \beta_{j_1}/\alpha_{j_1} = 0$ or $t_1 = (\beta_{j_1} - v_{j_1})/\alpha_{j_1} = 0$. Now the value of $|\alpha_{j_1}|$ is such that $r_2 = r_1 - |\alpha_{j_1}| < 0$, therefore $w(t) < w(0)$ for any $t > 0$, i.e., degeneracy and the magnitude of $|\alpha_{j_1}|$ prevent an increase in $w(t)$. This case carries the possibility of further variations that will be analyzed later.

When attempting to maximize $w(t)$, condition 1 has to be observed. It stipulates that the incoming variable takes a feasible value. If the incoming x_q is a type-1 variable it may happen that the t_k that maximizes $w(t)$, is greater than or equal to its own upper bound u_q , i.e., $t_k \geq u_q$. In this case it is worth stopping at $t = u_q$ and perform a bound flip as it is a cheap improving iteration (no basis change).

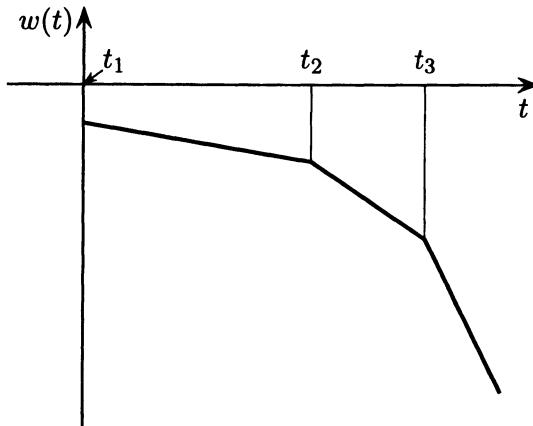


Figure 9.9. Special degenerate case ($t_1 = 0$) when $w(t)$ has no ascending part.

From the above discussion we can conclude the following.

PROPOSITION 9.6.1 *If we are not faced with a bound flip then $w(t)$ attains its maximum at a break point $t = t_s$. This break point defines a basis change where both the incoming and outgoing variables take a feasible value.*

Having determined the t_k break points we can easily find the global maximum of $w(t)$ since it is reached at a point where the sign of the slope of $w(t)$ changes. This is stated formally in the following proposition.

PROPOSITION 9.6.2 *Let $r_1 = -d_q$ the initial slope of $w(t)$ and compute*

$$r_{k+1} = r_k - |\alpha_{j_k}|, \quad k = 1, 2, \dots \quad (9.90)$$

The maximum of $w(t)$ is defined by index s for which

$$r_s > 0 \quad \text{and} \quad r_{s+1} \leq 0 \quad (9.91)$$

hold. If the slope is zero between two break points they both maximize $w(t)$.

In some cases it may be useful to find the ‘farther’ break point which can be achieved if the (9.91) conditions are replaced by

$$r_s \geq 0 \quad \text{and} \quad r_{s+1} < 0. \quad (9.92)$$

In this chapter, s will be used to denote the index of the break point in the sorted order that defines the maximum of $w(t)$ in the sense of the proposition.

The optimum value of $w(t)$ can be obtained from its definition (9.86) by substituting t_s for t as $w(t_s)$. This value can also be computed by a recursion done in parallel with the calculation of the slopes as shown in the next proposition.

PROPOSITION 9.6.3 *Let $t_0 = 0$ and compute the*

$$w(t_k) = w(t_{k-1}) + (t_k - t_{k-1})r_k, \quad k = 1, \dots, s. \quad (9.93)$$

recursion. The maximum of $w(t)$ is the s -th iterate $w(t_s)$.

The statement can be verified by using Propositions 9.6.1 and 9.6.2 and the definition of t_k and r_k . Figure 9.7 can be used to visualize the proof.

If the displacement of the incoming variable is negative (the $t \leq 0$ case) we can simply use $-\alpha_i$ instead of α_i in the previous discussion and everything remains valid.

The above described procedure is known as *FEWPHI* referring to the fact that it possibly makes few iterations in phase-1 compared with other methods.

Note that if we take $s = 1$, i.e., the smallest ratio, we obtain the traditional method for determining the outgoing variable. In this respect the above described method is a generalization of the traditional phase-1 procedure that can handle infeasible basic variables. It entails that any time when we have $s > 1$ a locally stronger iteration is made. It can lead to a dramatic decrease in the number of iterations in phase-1 but, obviously, there is no guarantee for that. However, computational evidence strongly supports its superiority in a vast majority of cases which makes it an ideal choice for primal phase-1. The number of infeasibilities can temporarily go up but it also can go down substantially in one step. Unfortunately, type-0 variables can usually be made feasible only one per iteration.

EXAMPLE 9.5 Assume, an infeasible basic solution, $\beta = \mathbf{x}_B$, a type-2 nonbasic variable at lower bound ($=0$) and its transformed column α are given. First, evaluate feasibility of β , compute w , verify that the chosen variable is an improving candidate (compute phase-1 reduced cost) then determine the ratios, the outgoing variable (if any), the displacement and value of the incoming variable, the new solution and the new w .

The left part of the table below, up to the double vertical lines, contains the given data. To the right of it the feasibility status of the basic variables and the computed ratios are shown.

i	$type(\beta_i)$	β_i	v_i	α_i	Feasibility	τ_{ℓ_i}	τ_{u_i}
1	0	0	0	-1	\mathcal{F}	—	0
2	1	4	1	1	\mathcal{P}	4	3
3	2	-25	∞	-5	\mathcal{M}	5	—
4	0	2	0	2	\mathcal{P}	1	1
5	1	2	4	-1	\mathcal{F}	—	2
6	2	-1	∞	-1	\mathcal{M}	1	—

\mathcal{F} , \mathcal{M} and \mathcal{P} denote the feasibility status, see (9.9), (9.10) and (9.11). The negative of the sum of the infeasibilities is w as defined in (9.76). In this case it is $w = (-25 - 1) - ((4 - 1) + 2) = -31$.

The phase-1 reduced cost of the variable can be determined by (9.81) as $d = (-5 - 1) - (1 + 2) = -9$. Therefore, the improving displacement is $t \geq 0$ which is feasible for the incoming variable. Thus, it is, indeed, an improving candidate.

The above table shows that two basic variables define two break points each while the others just one. Now, the break points have to be sorted in an ascending order and also the type of the break point (τ_{ℓ_i} or τ_{u_i}) has to be remembered. This latter can be achieved, for example, by reversing the sign of the row index for τ_{u_i} and taking its absolute value when used for indexing. After sorting the break points we set up the following table.

k	t_k	j_k	α_{j_k}
1	0	-1	-1
2	1	4	2
3	1	-4	2
4	1	6	-1
5	2	-5	-1
6	3	-2	1
7	4	2	1
8	5	3	-5

Now we can apply the iterative procedures of Propositions 9.6.2 and 9.6.3 to determine the break point that maximizes $w(t)$ and also the maximum. By definition, $t_0 = 0$ and $w(0) = w$.

k	r_k	α_{j_k}	t_k	$w(t_k)$	j_k	v_{j_k}
0	—	—	0	-31	—	—
1	9	-1	0	-31	-1	0
2	8	2	1	-23	4	0
3	6	2	1	-23	-4	0
4	4	-1	1	-23	6	$+\infty$
5	3	-1	2	-20	-5	4
6	2	1	3	-18	-2	1
7	1	1	4	-17	2	1
8	0	-5	5	-17	3	$+\infty$
9	—	—	—	—	—	—

If stopping criterion (9.91) is used then $s = 7$ (the 7th break point) and $t_s = 4$ are obtained as the maximizer with $\beta_{|j_7|} \equiv \beta_2$ leaving at lower bound (indicated by $j_7 > 0$). Using (9.92) results in $s = 8$ and $t_s = 5$ ($\beta_{|j_8|} \equiv \beta_3$ leaving at lower bound). In both cases $w(t_s) = -17$ though the solutions are different as shown below. (Note, the incoming variable is not bounded from above.)

Solution with $t_s = 4$:

$$\beta(4) = \beta - 4 \times \alpha = \begin{bmatrix} 0 \\ 4 \\ -25 \\ 2 \\ 2 \\ -1 \end{bmatrix} - 4 \times \begin{bmatrix} -1 \\ 1 \\ -5 \\ 2 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \\ -5 \\ -6 \\ 6 \\ 3 \end{bmatrix}.$$

In this case variable $\beta_{|j_7|} \equiv \beta_2$ leaves the basis and the incoming variable takes its position as $\bar{\beta}_2 = 0 + t_7 = 4$.

If the index set of basic variables is $\mathcal{B} = \{k_1, k_2, k_3, k_4, k_5, k_6\}$ and the subscript of the incoming variable is q then the new basis is $\bar{\mathcal{B}} = \{k_1, q, k_3, k_4, k_5, k_6\}$ and the new solution is $\bar{\beta} = \mathbf{x}_{\bar{\mathcal{B}}} = [4, 4, -5, -6, 6, 3]^T$. The measure of infeasibilities is $\bar{w} = w(t_7) = w(4) = -17$. This value can be verified directly from the new solution.

The other solution is obtained with $t_8 = 5$:

$$\beta(5) = \beta - 5 \times \alpha = \begin{bmatrix} 0 \\ 4 \\ -25 \\ 2 \\ 2 \\ -1 \end{bmatrix} - 5 \times \begin{bmatrix} -1 \\ 1 \\ -5 \\ 2 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 5 \\ -1 \\ 0 \\ -8 \\ 7 \\ 4 \end{bmatrix}.$$

Now $\beta_{|j_8|} \equiv \beta_3$ leaves the basis at lower bound and the incoming variable takes its position, $\bar{\beta}_3 = 0 + t_8 = 5$.

In a similar vein as above, the new basis is $\bar{\mathcal{B}} = \{k_1, k_2, q, k_4, k_5, k_6\}$ and the new solution is $\bar{\beta} = \mathbf{x}_{\bar{\mathcal{B}}} = [5, -1, 5, -8, 7, 4]^T$. The measure of infeasibilities is again $\bar{w} = w(t_8) = w(5) = -17$.

It is instructive to see how the infeasibility structure of the basis changes in either case. Namely, some infeasible positions become feasible and vice versa.

9.6.2 The logic of the ratio test

The logic of the phase-1 ratio test is summarized in a pseudo code in Algorithm 2 on page 218. It contains only the generation of break points. To make use of them they have to be sorted into an ascending order so that the maximization of $w(t)$ can be carried out by the recursions described in Propositions 9.6.2 and 9.6.3. As the number of break points can be large sorting has to be organized carefully. More details are discussed under computational considerations in section 9.6.4 where some important aspects of its implementation are also presented.

9.6.3 Extensions to the ratio test

In addition to allowing feasible basic variables to become infeasible (relaxation of condition 2) we can investigate whether any advantage can be gained if the incoming variable is also allowed to enter the basis at an infeasible level (relaxation of condition 1).

The incoming variable can be infeasible in two ways:

- with a value greater than its upper bound,
- with a negative value of a non-free variable.

Type-3 variables are excluded from the further discussion as they enter the basis always at a feasible level. For the remaining variables the following example can help highlight the situation.

Algorithm 2 Primal Phase-1 Ratio Test

Require: An infeasible basis with \mathcal{B} , \mathcal{U} and basic solution $\beta = \mathbf{x}_{\mathcal{B}}$,
 Phase-1 reduced cost d_q of the incoming candidate and its updated
 column $\alpha = \alpha_q$. The upper bound vector of basic variables is de-
 noted by v .

Ensure: Number S and unsorted list t_1, \dots, t_S of break points and
 signed list j_1, \dots, j_S with their indices and types.

```

1: Initialize:  $S := 0$ 
2:  $\sigma := 1$  {Signifies the  $t \geq 0$  case}
3: if  $d_q > 0$  then
4:    $\sigma := -1$  {Signifies  $t \leq 0$ }
5: end if
6: for  $i := 1$  to  $m$  do
7:   if  $\alpha_i \neq 0 \wedge \text{type}(\beta_i) \neq 3$  then {type-3 basic variables ignored}
8:      $a := \sigma \times \alpha_i$ ;  $k := 0$ 
9:     if  $a < 0$  then { $\beta_i(t)$  is increasing}
10:      if  $i \in (\mathcal{F} \cup \mathcal{M}) \cap \mathcal{I}_u$  then { $\beta_i$  has finite upper bound}
11:         $k := 2$  { $\tau_{u_i}$  is defined}
12:      end if
13:      if  $i \in \mathcal{M}$  then
14:         $k := k + 1$  { $\tau_{\ell_i}$  is defined}
15:      end if
16:    else { $\beta_i(t)$  is decreasing}
17:      if  $i \in \mathcal{F} \cup \mathcal{P}$  then
18:         $k := 1$  { $\tau_{\ell_i}$  is defined}
19:        if  $i \in \mathcal{P}$  then
20:           $k := 3$  { $\tau_{u_i}$  is also defined}
21:        end if
22:      end if
23:    end if
24:    if  $k \neq 0$  then
25:      if  $k$  is odd then {Compute and store  $\tau_{\ell_i}$ }
26:         $S := S + 1$ ;  $t_S := \beta_i/a$ 
27:         $j_S := i$  {Signifies  $\tau_{\ell_i}$ }
28:      end if
29:      if  $k > 1$  then {Compute and store  $\tau_{u_i}$ }
30:         $S := S + 1$ ;  $t_S := (\beta_i - v_i)/a$ 
31:         $j_S := -i$  {Signifies  $\tau_{u_i}$ }
32:      end if
33:    end if
34:  end if
35: end for

```

EXAMPLE 9.6 Assume again, an infeasible basic solution, $\beta = \mathbf{x}_B$, and the transformed column α are given. Now the improving candidate is at its upper bound of 4. Evaluate feasibility of β , compute w , verify that the chosen variable is an improving candidate then determine the ratios, the outgoing variable (if any), the displacement and value of the improving candidate, the new solution and the new w .

As in Example 9.5, the left part of the table below, up to the double vertical lines, contains the given data. To the right of it the feasibility status of the basic variables and the computed ratios are shown. Note that by taking $-\alpha_i$ we compute $-\tau_{\ell_i}$ or $-\tau_{u_i}$. Otherwise, notations are the same as before.

i	$type(\beta_i)$	β_i	v_i	α_i	Feasibility	$-\tau_{\ell_i}$	$-\tau_{u_i}$
1	1	6	2	-2	\mathcal{P}	3	2
2	1	3	4	1	\mathcal{F}	-	1
3	2	3	∞	4	\mathcal{F}	-	-
4	2	-30	∞	6	\mathcal{M}	5	-
5	1	8	2	1	\mathcal{P}	-	-

The measure of infeasibilities is $w = -30 - (4 + 6) = -40$.

The phase-1 reduced cost of the variable is $d = 6 - (-2 + 1) = 7$ which shows it is, indeed, an improving candidate as we are faced with the $t \leq 0$ case.

According to the above table four break points are defined. After sorting them we obtain:

k	$-t_k$	j_k	α_{j_k}
1	1	2	1
2	2	-1	-2
3	3	1	-2
4	5	4	6

The step-by-step application of the iterative procedures of Propositions 9.6.2 and 9.6.3 for determining the maximizing point results in the following table.

k	r_k	α_{j_k}	$-t_k$	$w(t_k)$	j_k	v_{j_k}
0	—	—	0	-40	—	—
1	7	1	1	-33	2	4
2	6	-2	2	-27	1	2
3	4	-2	3	-23	1	2
4	2	6	5	-19	4	$+\infty$
5	-4					

Either stopping rule gives $s = 4$ (the maximum of $w(t)$ is unique) with $-t_s = 5$, i.e., a displacement of -5 of the incoming variable and $w(-5) = -19$. However, as the variable is coming in from its upper bound of 4, this displacement would make it infeasible. Thus, to satisfy condition 1 of phase-1 iterations, it is set to its opposite bound (0 in this case, which corresponds to $t = -4$) resulting in an iteration without basis change. The new solution is obtained by substituting -4 for t in the expression of $\beta(t)$:

$$\beta(-4) = \beta - (-4) \times \alpha = \begin{bmatrix} 6 \\ 3 \\ 3 \\ -30 \\ 8 \end{bmatrix} + 4 \times \begin{bmatrix} -2 \\ 1 \\ 4 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 7 \\ 19 \\ -6 \\ 12 \end{bmatrix}.$$

Evaluating the infeasibility of this solution (either directly or by substituting -4 into $w(t)$) we obtain -21.

There is an interesting situation here. If we allow the incoming variable to travel $t = -5$ it becomes infeasible and contributes to the sum of infeasibilities by -1. On the other hand, the infeasibility of the basis reduces to -19. Therefore, if the incoming variable replaces β_4 at a level of -1 the new basic solution is still less infeasible than the one obtained by a bound flip of the incoming candidate.

Assume a nonbasic variable x_j is at zero, its upper bound is u_j and we want to move it away from zero. Denote $W_j(t)$ the infeasibility function in a similar sense as in (9.86). It is easy to see that the measure of infeasibility has the following form now:

$$W_j(t) = \sum_{i \in \mathcal{I}_e} [\beta_i - t\alpha_j^i]^- - \sum_{i \in \mathcal{I}_u} [\beta_i - t\alpha_j^i - u_j]^+ + g_j(t),$$

where

$$g_j(t) = \begin{cases} t & \text{if } t < 0, \\ u_j - t & \text{if } t > u_j, \\ 0 & \text{otherwise.} \end{cases} \quad (9.94)$$

Obviously, $W_j(t) \leq 0$. If it is zero the corresponding solution is feasible. Using the definition of $w(t)$ in (9.86) with explicitly indicating its dependency on index j , $W_j(t)$ can be rewritten as

$$W_j(t) = w_j(t) + g_j(t).$$

Moving x_j away from its current nonbasic level, the measure of infeasibility, $W_j(t)$, can be improved if one of the conditions of the following lemma is satisfied.

LEMMA 9.2 *Assume x_j is a nonbasic variable at zero and its displacement is denoted by t . $W_j(t)$ can be improved by changing the value of x_j*

I. by $t \geq 0$ if

$$d_j = \sum_{i \in M} \alpha_j^i - \sum_{i \in P} \alpha_j^i < 0, \quad (9.95)$$

or

II. by $t \leq 0$ if

$$d_j = \sum_{i \in M} \alpha_j^i - \sum_{i \in P} \alpha_j^i > 1. \quad (9.96)$$

PROOF. In case I. we have $t \geq 0$. As $g_j(t) = 0$ for $0 \leq t \leq u_j$, $W_j(t)$ reduces to $w_j(t)$ (which is $w(t)$ of (9.86)) and so does (9.95) to lemma 9.1

In case II. the displacement is negative. The change in $W_j(t)$ is

$$\Delta W_j = W_j(t) - W_j(0) = -td_j + \Delta g_j = -td_j + g_j(t), \quad (9.97)$$

as $g_j(0) = 0$. Since $g_j(t) = t$ for $t \leq 0$, (9.97) can be rewritten as

$$\Delta W_j = -td_j + t.$$

Here, $d_j > 1$ is needed for $\Delta W_j > 0$ which proves case II. \square

The opposite possibility is highlighted in the following lemma which is given without the obvious proof.

LEMMA 9.3 *Assume x_j is a nonbasic variable at its upper bound of u_j . Its displacement is denoted by t . $W_j(t)$ can be improved by changing the value of x_j*

I. by $t \leq 0$ if

$$d_j = \sum_{i \in \mathcal{M}} \alpha_j^i - \sum_{i \in \mathcal{P}} \alpha_j^i > 0,$$

or

II. by $t \geq 0$ if

$$d_j = \sum_{i \in \mathcal{M}} \alpha_j^i - \sum_{i \in \mathcal{P}} \alpha_j^i < -1.$$

From (9.94) it is easy to see that $g_j(t)$ is a piecewise linear concave function for $-\infty < t < +\infty$. Therefore, $W_j(t)$ is also such. It has the same break points as $w_j(t)$ has but an additional break point may also be defined by $g_j(t)$. How the latter can happen is summarized in the following table. Zero subscript is used to distinguish such a break point from the others.

	$u_j < +\infty$		$u_j = +\infty$
	$x_j = 0$	$x_j = u_j$	$x_j = 0$
$t \geq 0$	$d_j < 0$ $\tau_{u_0} = u_j$	$d_j < -1$ $\tau_{u_0} = 0$	$d_j < 0$ —
$t \leq 0$	$d_j > 1$ $\tau_{\ell_0} = 0$	$d_j > 0$ $\tau_{\ell_0} = -u_j$	$d_j > 1$ $\tau_{\ell_0} = 0$

(9.98)

Every time when a τ_{ℓ_0} or a τ_{u_0} is defined the corresponding $\alpha_0 = 1$ as the incoming variable starts contributing to the sum of infeasibilities at a rate of 1 after it goes beyond the break point.

Returning to the situation of lemma 9.2, namely, when $x_j = 0$, two different sets of break points (including the one from (9.98)) can be generated corresponding to cases I and II. They need to be sorted into the following order:

$$0 \leq t_1 \leq \dots \leq t_{S_1} \quad (9.99)$$

and

$$t_{S_2} \leq \dots \leq t_1 = 0. \quad (9.100)$$

For notational simplicity we drop subscript j from $W_j(t)$ and refer to it as $W(t)$ in the sequel.

Now a statement analogue to Proposition 9.6.2 is the following.

PROPOSITION 9.6.4 *Assume break points of $W(t)$ have been determined and sorted as in (9.99) or (9.100). If we set $r_1 = -d_j$ then the slope of the $(k+1)$ st interval of $W(t)$ is in case I*

$$r_{k+1} = r_k - |\alpha_{j_k}|, \quad k = 1, \dots, S_1,$$

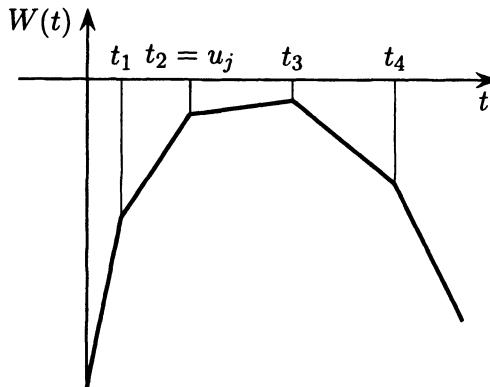


Figure 9.10. Typical shape of $W(t)$ with $d_j < 0$, $t \geq 0$ and u_j finite.

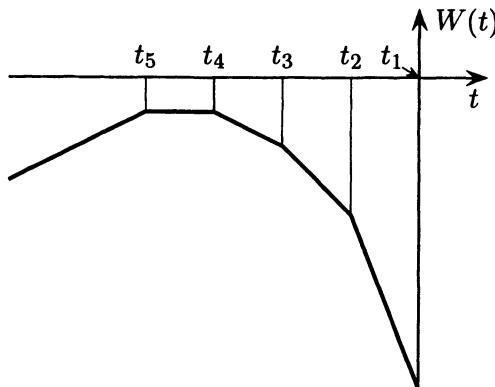


Figure 9.11. Typical shape of $W(t)$ with $d_j > 1$ and $t \leq 0$. Note the horizontal section between t_4 and t_5 : t_4 and t_5 represent alternative optima.

and in case II

$$r_{k+1} = r_k + |\alpha_{j_k}|, \quad k = 1, \dots, S_2.$$

PROOF. The proof is based on arguments similar to those used in Proposition 9.6.2 and can be reconstructed accordingly. \square

In case I the shape of $W(t)$ is similar to the basic patterns of $w(t)$ in figures 9.7, 9.8 and 9.9, while in case II $W(t)$ is the reflection of $w(t)$ to the vertical axis in such a way that $t_1 = 0$ is always true. Typical shapes are shown in Figures 9.10 and 9.11.

Finding the maximum of $W(t)$ can be done in an analogous way shown in Proposition 9.6.2. It is instructive to note how the search for the maximum of $W(t)$ can go beyond the break point defined by the upper

bound of the incoming variable, i.e., $t_k = u_j$. The k -th break point is preceded by the k -th linear interval with the slope of r_k . Since now $\alpha_{i_k} = 1$ the slope after this break point is

$$r_{k+1} = r_k - 1. \quad (9.101)$$

If $r_{k+1} > 0$ then the stopping criterion is not satisfied and the maximum of $W_j(t)$ is not achieved. From (9.101), $r_{k+1} > 0$ is equivalent to

$$r_k > 1. \quad (9.102)$$

In other words, if the slope of $W(t)$ does not fall below 1 until $t = u_j$ then the maximum of $W(t)$ is achieved at an infeasible value of the incoming variable.

It is worth noticing the analogy between (9.96) and (9.102), i.e., the conditions for the incoming variable to enter the basis at an infeasible level.

Experience shows that in the early stages of phase-1, especially when there are many infeasible variables anyhow, it is usually beneficial to let the incoming variable take an infeasible value if it results in a better reduction of the measure of infeasibility. In other words, the use of $W(t)$ seems justified. Closer to feasibility, the creation of a new infeasibility is usually not advantageous and the use of $w(t)$ is generally better in terms of the total number of phase-1 iterations.

9.6.4 Computational considerations

Determining the outgoing variable in phase-1 based on $w(t)$ or $W(t)$ results in a locally strong iteration as the largest possible step is made towards feasibility. Some additional features and requirements of this method are also of interest.

9.6.4.1 Coping with degeneracy

Recalling (9.12), a basis is degenerate if $\mathcal{D} \neq \emptyset$, i.e., if at least one of the basic variable is at one of its bound. The danger of a degeneracy is that if a basic variables is at bound with the ‘wrong’ sign of α_i it can block the displacement of the incoming variable ($t = 0$) thus prohibiting a progress towards feasibility. Such a blocking can particularly easily occur with traditional phase-1 methods that satisfy condition 2 of the basis change.

The situation is different if this condition is relaxed and $w(t)$ or $W(t)$ is used to determine the outgoing variable. If the basis is degenerate the following break point may be defined (in the $t \geq 0$ case)

$$0 = t_1 = \cdots = t_\ell < t_{\ell+1} \leq \cdots \leq t_S. \quad (9.103)$$

According to Proposition 9.6.2 the maximum of $w(t)$ is defined by index s of the break points for which

$$r_s = r_1 - \sum_{k=1}^{s-1} |\alpha_{j_k}| > 0 \quad \text{and} \quad r_{s+1} = r_1 - \sum_{k=1}^s |\alpha_{j_k}| \leq 0,$$

or, taking into account that $r_1 = -d_j$,

$$\sum_{k=1}^{s-1} |\alpha_{j_k}| < -d_j \quad \text{and} \quad \sum_{k=1}^s |\alpha_{j_k}| \geq -d_j.$$

If for this s relation $s > \ell$ holds then, by (9.103), $t_s > 0$. Therefore, despite the presence of degeneracy a positive step can be made towards feasibility. Obviously, such a step may result in an increase of the number of infeasibilities but the measure will definitely decrease. As a side effect the creation of new infeasibilities can lead to subsequent non-degenerate iterations.

If $s \leq \ell$ then $t_s = 0$. Thus, degeneracy prevents an improvement of $w(t)$. In this case the incoming variable enters the basis with zero displacement.

9.6.4.2 Numerical stability

One of the reasons for the occasional numerical troubles in the simplex method is the improper (which usually means ‘very small’) size of some pivot elements. In case of the traditional method it is hard to do anything about it as there is no choice if the minimum ratio is unique. If it is not unique a ratio with a better-sized pivot element can be chosen.

Using $w(t)$ or $W(t)$ a large flexibility is created if $s > 1$, i.e., the maximizer is not the first break point. If the pivot corresponding to t_s is too small then—usually losing optimality of $w(t)$ or $W(t)$ —we can go one step back and consider the pivot for $s := s - 1$. This procedure can be repeated if necessary until $s = 1$. On the other hand, it is possible to go beyond s in the course of the search for a proper pivot element giving $s := s + 1$ as long as $s \leq S$ and $w(t_s) \geq w(0)$.

9.6.4.3 Multiple pricing

If multiple pricing is used the incoming vector is determined from among the selected candidates based on the largest improvement. It requires the performance of ratio tests for each candidate. In case of the traditional phase-1 method ($s = 1$ is chosen) the progress of the objective function can directly be computed. If $w(t)$ or $W(t)$ is used then, in addition to the ratio test, the actual progress with each candidate has to be computed by (9.93) to be able to select the most favorable one.

9.6.4.4 Memory requirements

FEWPHI requires the storage of the calculated τ_{ℓ_i} and τ_{u_i} values. Since each basic variable can define at most 2 break points their maximum number is $2m$, i.e., $S \leq 2m$. Additionally, a permutation vector (j_1, \dots, j_S) is also needed to remember which row (basic variable) defined which t_k value. Whether a t_k represents a τ_{ℓ_i} or τ_{u_i} can be indicated by the sign of j_k as shown in the examples.

9.6.4.5 Computational effort

Computing the τ_{ℓ_i} and τ_{u_i} threshold values according to (9.87) and (9.88) does not involve much extra work as most of these quantities are computed when the traditional method is used. The main difference is that these values need to be stored. They also have to be sorted as needed for the direct execution of (9.90) and (9.93) to determine the maximum of $w(t)$. Sorting all S items may be computationally very expensive if S is large (say, over 100). However, in most cases not all of them are used during the maximization. In fact, in step k of the recursive forms only the k -th smallest t is needed. Therefore, the best practice is to use a sorting algorithm which in step k gives the sorted order of the first k smallest ratios. Simultaneously with step k of the sorting, step k of recursions (9.90) and (9.93) can also be computed. When the maximum of $w(t)$ is found there is no need to sort the remaining t_i , $i = s+1, \dots, S$ values.

If we knew in advance that only a few break points will be needed out of arbitrarily many (i.e., s is going to be small) *selection sort* or even *bubble sort* could be used as these algorithms satisfy the above requirement and have very little overhead. However, as their complexity is $O(s^2)$ they become prohibitively slow if s is large which is not uncommon if S is large. A very fast and predictable sort algorithm is *heap sort* which is usually discussed under the heading *priority queues*. Its overhead for setting up the heap is linear in S and retrieves the first k smallest items in no more than $O(k \log_2 k)$ elementary steps.

Experience shows that the number of break points can change dramatically from one iteration to the next. For example, S can go to the thousands and in a subsequent iteration can drop under 10, then it may soar to the hundreds or thousands again. Additionally, the number of break points s used for maximization also shows quite erratic fluctuations, mostly in the early stages of phase-1 iterations. It is not uncommon that s is in the hundreds or thousands, reaching 50–100% of S . However, it is more typical that $1 \leq s \ll S$. As phase-1 progresses S shows a strong downturn tendency and s tends to settle under 5, even-

tually staying at 1. In the latter case FEWPHI acts as a traditional phase-1 method.

9.6.5 Adaptive composite pricing in phase-1

During phase-1 of the simplex method the true objective function $z = \mathbf{c}^T \mathbf{x}$ is completely ignored. It means that the first feasible solution (if one exists) can be any far from an optimal solution. As a consequence, a large number of phase-2 iterations may be needed to solve the problem.

A rather straightforward idea is to give some consideration to the true objective during the search for feasibility. This problem can be viewed as a multi-objective optimization problem with two objectives: one measures the w level of infeasibility and the other is the true $z = \mathbf{c}^T \mathbf{x}$ objective function value.

A simple attempt to address the problem is to combine the two objectives into a *composite objective function*

$$f = w - \gamma z, \quad (9.104)$$

with $\gamma \geq 0$ and maximize it subject to $\mathbf{Ax} = \mathbf{b}$ and the type constraints. Recall that the (9.77) – (9.79) problem is a maximization while the original problem is a minimization. In this criterion the true objective is represented with a nonnegative weight γ which is a (user definable) parameter. It is usually a small value between 0.01 and 0.1. In practical terms (9.104) means that we look for improving candidates for f . If such a variable does not exist we are at an f -optimal solution. If $w = 0$ at this point, the corresponding \mathbf{x}_B is an optimal solution to the original problem. However, if $w < 0$ we cannot decide on the feasibility of the problem. In this case we set $\gamma = 0$ and make a pure phase-1.

If the problem has no feasible solution then an unfortunate choice of the value of γ can seriously delay the identification of infeasibility. This is because there can be candidates that are detrimental to w but are favorable for z and, due to the relative magnitudes of w , γ and z , they still can improve f .

Of course, the problem can have a feasible solution despite the $w < 0$ condition at the f -optimal solution. In this case, during the ensuing pure phase-1, all gain in z can be lost. Even worse than that, the whole composite optimization might be just waste of time. In practice, there is very little success with this approach.

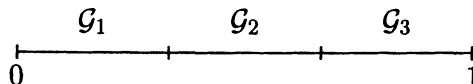
A possible improvement of this procedure is the dynamic adjustment of γ . While this can lead to a somewhat better performance it still does not address the main problem outlined above.

We can try a different way of working with two objectives. As in phase-1 feasibility is more important than optimality, we can view the

two objectives in a hierarchy. Therefore, a candidate is considered only if it is profitable for w . From among these variables we make a choice based on the evaluation of the composite objective function in (9.104). Additionally, we can dynamically change the value of γ by an adaptive algorithm. If there are no more improving candidates for w then phase-1 is terminated. Now we can say with certainty that if $w < 0$ then the problem is infeasible. Otherwise, a BFS has been found (with a hopefully good z) and phase-2 can start.

These ideas have been described more formally by Maros in [Maros, 1981, Maros, 1986] where the resulting algorithm is called ADACOMP (ADAptive COMposite Procedure). Below we give a summary of the traditional composite approach and a bit more detailed description the two adaptive procedures.

Let z_j denote the phase-2 reduced cost of x_j (the same as d_j in (2.27)) and w_j the phase-1 reduced cost (the same as d_j in (9.81)). It is easy to see that the reduced cost of the composite objective is $f_j = z_j - \gamma w_j$. We divide the $\mathcal{G} = [0, 1]$ interval into three disjunct subintervals $\mathcal{G}_1, \mathcal{G}_2$ and \mathcal{G}_3 such that $\mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2 \cup \mathcal{G}_3$:



As ADACOMP is often combined with multiple pricing we give its description in this environment with maximum H vectors selected in a major iteration. Obviously, by setting $H = 1$ we get the single pricing version of it.

As long as there are no more than H improving candidates according to z_j they are all selected. If more are found a secondary selection criterion is used which favors those vectors for which $z_j - \gamma w_j$ is more negative (or positive, depending on the status of the nonbasic variable). Therefore, all variables in the list are improving for w and not very harmful (or even improving) for z . If the selection is followed by a suboptimization the minor iterations are continued as long as the candidates are still improving for w .

To dynamically adjust γ , two counts are used during a major iteration. ν_w counts the number of improving candidates for w and ν_f counts those which, additionally, are improving for f .

If $\nu_w = 0$ then the problem is infeasible (the $w < 0$ case) independent of the current value of γ , or a feasible solution has been found ($w = 0$).

If $\nu_w > 0$ then the value of γ is modified for the next major iteration in the following way. Let $\varrho = \nu_f / \nu_w$. Obviously, $0 \leq \varrho \leq 1$. The new

value of γ is determined by

$$\bar{\gamma} = \begin{cases} g_1(\gamma), & \text{if } \gamma \in \mathcal{G}_1, \\ \gamma, & \text{if } \gamma \in \mathcal{G}_1, \\ g_3(\gamma), & \text{if } \gamma \in \mathcal{G}_3. \end{cases} \quad (9.105)$$

Here, functions g_1 and g_3 satisfy the following conditions: $0 \leq \bar{\gamma} = g_1(\gamma) < \gamma$ and $0 \leq \gamma < \bar{\gamma} = g_3(\gamma)$.

In verbal terms (9.105) can be interpreted in the following way. If the w -improving variables are mostly improving for f too, $\varrho \in \mathcal{G}_3$, then this is a favorable situation and a larger weight can be given to the true objective function which is achieved by increasing γ . In the opposite case, $\varrho \in \mathcal{G}_1$, many of the w -improving variables are not favorable for f which suggests conflicting interests. However, feasibility is considered more important which can be emphasized by decreasing the weight of z . If $\varrho \in \mathcal{G}_2$ we can say that the two components of f are in a—user defined—balance, there is no need to change γ .

This procedure has several parameters that we can ‘play with’. They are $\mathcal{G}_1, \mathcal{G}_2$ and \mathcal{G}_3 , $g_1(\gamma), g_3(\gamma)$ and the initial value of γ . Acceptable defaults are

$$\mathcal{G}_1 = [0, 1/3], \mathcal{G}_2 = [1/3, 2/3], \text{ and } \mathcal{G}_3 = [2/3, 1].$$

$$g_1(\gamma) = \gamma/2, \quad g_3(\gamma) = 2\gamma.$$

$$\text{Initial value } \gamma = 0.05.$$

The use of ADACOMP does not come for free. First of all we have to determine the phase-1 simplex multiplier ϕ as in (9.83) and also the phase-2 simplex multiplier π , see (2.28). The former is a regular phase-1 procedure the latter is an extra in phase-1. If, during the scanning of nonbasic variables, a w -improving vector is found its f_j has to be determined by a further dot product (to obtain z_j). The summary of the extra work required by ADACOMP is a BTRAN to compute π and an additional dot product with the w -improving vectors.

ADACOMP opens up a large flexibility which can be beneficial for the simplex method. It has shown some remarkable performance improvements. However, its sensitivity to the choice of the parameters makes it difficult to customize.

A simplified version of ADACOMP can be obtained if we are satisfied with an automatic readjustment of γ . In this approach we want to maintain a predefined relative balance between the two components of f . For instance, we can say that the relative weight of w should be always $\varphi = 0.90$. In general, a prescribed ratio φ can be maintained by

adjusting γ according to the actual values of w and z . If $z \neq 0$ then from the required

$$\frac{|w|}{\gamma|z|} = \varphi$$

ratio γ can be determined as

$$\gamma = \frac{|w|}{\varphi|z|}. \quad (9.106)$$

In practice we need to request that $|z|$ is greater than a reasonable tolerance. This condition is almost always automatically satisfied.

This method works very much in the same way as ADACOMP but γ is adjusted after every (major) iteration according to (9.106). The computational effort of using this modified version is practically the same as with ADACOMP. Again, this method has shown some beneficial effects on the overall solution time of several problems but does not display a uniform improvement.

In an advanced implementation of the simplex method it is necessary to include a composite phase-1 method which can be arbitrarily activated. Currently there is not enough computational evidence to say which version is the best candidate for this purpose.

9.7. Handling degeneracy

While degeneracy carries the danger of cycling there are hardly any real world problems that display this undesirable phenomenon. This is due to several factors like (a) floating point computations create small numerical errors that actually serve as perturbation that help avoid zero steps, (b) modern solvers use sophisticated pricing strategies, (c) some pivot strategies (like FEWPHI) have natural anti-degeneracy properties. While cycling does not occur, there are many highly degenerate problems in practice where the simplex algorithm gets entangled in very long sequences of nonimproving iterations which is known as *stalling*. Getting through the labyrinth of the degeneracy graph at a degenerate vertex can be as complex as solving an LP problem as pointed out by Gal [Gal, 1990].

There are theoretically proven methods that guarantee the finite termination of the simplex method in the presence of degeneracy. In general, they are computationally not efficient because they either require an enormous amount of work per iteration (Charnes [Charnes, 1952], Dantzig et al. [Dantzig et al., 1955]) or lead to a very large number of—otherwise cheap—iterations (Bland's rule [Bland, 1977]). There is, however a method that is theoretically sound (finite termination) and re-

quires relatively little extra computations. It is Wolfe's 'ad hoc' method [Wolfe, 1963], also investigated by Ryan and Osborne [Ryan and Osborne, 1988]. This method has only one practical difficulty, namely it requires the identification of the degenerate positions (it is a 'degeneracy aware' procedure). This identification is simple if a basic variable is at one of its bounds. However, ambiguity arises when a computed value of a basic variable is very close to a bound. To resolve the problem degeneracy has to be identified with a tolerance. As primal and dual feasibility are also considered with a tolerance the procedure of identifying degeneracy in a similar way does not introduce further complications.

We redefine the degeneracy set of the basic variables to include the *near degenerate* positions. It is important to remember that type-3 basic variables are always nondegenerate even if they are equal to zero (and they do not block any incoming variable). Therefore, they do not take part in issues related to degeneracy. The index set of the (near) degenerate positions is denoted by \mathcal{D} which defined as the union of two subsets \mathcal{D}^+ and \mathcal{D}^- .

$$\mathcal{D}^+ = \{i : |\beta_i| \leq \varepsilon_d\} \quad (9.107)$$

and

$$\mathcal{D}^- = \{i : |\beta_i - v_i| \leq \varepsilon_d \max\{1, |v_i|\}\}. \quad (9.108)$$

ε_d is the *degeneracy tolerance* which can take the value of the relative feasibility tolerance. It is to be noted that a degenerate type-0 basic variable belongs to both sets. If explicit lower bounds are used then (9.107) changes to

$$\mathcal{D}^+ = \{i : |\beta_i| \leq \varepsilon_d \max\{1, |\lambda_i|\}\},$$

where λ_i is the lower bound of the i -th basic variable (which is implicitly zero in (9.107)). Degeneracy set \mathcal{D} is then defined as

$$\mathcal{D} = \mathcal{D}^+ \cup \mathcal{D}^-. \quad (9.109)$$

There are heuristic methods that have been introduced to overcome stalling. They have been designed either to make a better choice of the incoming variables (pricing) or to choose the outgoing variable (pivot row) in a way that leads, rather sooner than later, to an improvement. One of the sophisticated methods is the EXPAND procedure of Gill et al. [Gill et al., 1989] that has become quite popular as it does not require the identification of the degenerate positions ('degeneracy unaware' procedure) and has a good performance in most of the cases. At the same time, some simple methods like *perturbation* and *shifting* which are 'degeneracy aware' procedures also show outstanding effectiveness in getting out of a stalling.

In this section we summarize some practical considerations for anti-degenerate pricing then discuss Wolfe's method, give details of EXPAND followed by the description of perturbation and shifting.

9.7.1 Anti-degeneracy column selection

The primary purpose of pricing is to select improving candidates with respect to the actual (phase-1 or phase-2) objective function. If the basic solution is (nearly) degenerate then even if the reduced cost looks very promising the iteration may end up with a small or zero steplength 'resulting' in practically no improvement. This is caused by small or zero ratios. However, during pricing the coefficients of the updated columns are not known, therefore, we do not know whether the degenerate positions will take part in the ratio test or not.

Greenberg [Greenberg, 1978b], Maros [Maros, 1981] and Nazareth [Nazareth, 1987] give criteria that help eliminate the definitely 'wrong' candidates, thus increasing the chance of the selection of columns that enable nondegenerate iterations in phase-2. (The situation in phase-1 is somewhat different as basic variables are allowed to go infeasible, see section 9.6 and, in particular, 9.6.4.) We briefly summarize their approaches.

First of all, we can characterize incoming columns which are 'good' for coping with degeneracy. It is stated in the following Proposition.

PROPOSITION 9.7.1 *Let $\alpha_j = \mathbf{B}^{-1}\mathbf{a}_j$ the updated form of an improving candidate. Its entry into the basis leads to a strictly improving step if*

$$V_j = \sum_{i \in \mathcal{D}} |\alpha_j^i| = 0. \quad (9.110)$$

The proof of this proposition is simple. Condition (9.110) means that $\alpha_j^i = 0$ for all degenerate positions. As a consequence, no zero valued ratios are defined which ensures a progress in the current objective function.

The only problem with the proposition is that it is practically impossible to check whether (9.110) holds or not for a nonbasic variable as the α_j^i components are unknown during pricing. If we are satisfied with a rather rough approximation then $\hat{V}_j = |\sum_{i \in \mathcal{D}} \alpha_j^i|$ can be used instead of V_j . \hat{V}_j can be determined by some extra computation in the following way. Let \mathbf{v} be defined by

$$v_i = \begin{cases} 1, & \text{if } i \in \mathcal{D}, \\ 0, & \text{otherwise.} \end{cases}$$

Then $\sum_{i \in \mathcal{D}} \alpha_j^i = \mathbf{v}^T \boldsymbol{\alpha}_j$, which is further equal to $\mathbf{v}^T \mathbf{B}^{-1} \mathbf{a}_j$. If we define an additional pricing vector $\hat{\mathbf{v}}$ to be

$$\hat{\mathbf{v}}^T = \mathbf{v}^T \mathbf{B}^{-1}$$

then \hat{V} can be obtained as

$$\hat{V}_j = |\hat{\mathbf{v}}^T \mathbf{a}_j|, \quad (9.111)$$

i.e., a dot product with the column which otherwise is an improving candidate with respect to the current objective function. So the extra work is (a) a BTRAN operation with \mathbf{v} and (b) a dot product only with the profitable nonbasic columns.

With a certain amount of optimism we can believe that columns for which \hat{V}_j is smaller are more likely to enable a nondegenerate iteration than others with larger \hat{V}_j .

Instead of (9.110) it would be sufficient to check whether

$$\sum_{i \in \mathcal{D}^+} (\alpha_j^i)^- + \sum_{i \in \mathcal{D}^+} (\alpha_j^i)^+ = 0, \quad (9.112)$$

where superscripts $-$ and $+$ are used in the sense as defined in (9.84) and (9.85). (9.112) actually means that all α_j^i coefficients in the degenerate positions are either zero or have a ‘good sign’ thus not taking part in the ratio test. It also tells us that if a type-0 basic variable has an $\alpha_j^i \neq 0$ in phase-2 then it will make the steplength zero as such an i belongs to both \mathcal{D}^+ and \mathcal{D}^- and will make the sum in (9.112) nonzero. Unfortunately, checking (9.112) is just as hopeless as (9.110).

At this point we can try to take advantage of the following proposition.

PROPOSITION 9.7.2 *If*

$$T_j = \sum_{i \in \mathcal{D}^+} \alpha_j^i - \sum_{i \in \mathcal{D}^-} \alpha_j^i > 0 \quad (9.113)$$

then only a degenerate step can be made with column j in phase-2.

PROOF. The (9.113) sum can be positive if there is either an $i \in \mathcal{D}^+$ with $\alpha_j^i > 0$ or an $i \in \mathcal{D}^-$ with $\alpha_j^i < 0$. In both cases the corresponding basic variable starts moving outside its feasibility domain thus it will be a blocking variable with zero stepsize. \square

The good thing is that T_j can be computed exactly. Define vector $\hat{\mathbf{t}}$ with components

$$\hat{t}_i = \begin{cases} +1, & \text{if } i \in \mathcal{D}^+ \\ -1, & \text{if } i \in \mathcal{D}^- \\ 0, & \text{otherwise.} \end{cases}$$

With this $\hat{\mathbf{t}}$ we can compute $T_j = \hat{\mathbf{t}}^T \boldsymbol{\alpha}_j = \hat{\mathbf{t}}^T \mathbf{B}^{-1} \mathbf{a}_j$. If notation $\mathbf{t}^T = \hat{\mathbf{t}}^T \mathbf{B}^{-1}$ is used then T_j can be obtained by

$$T_j = \mathbf{t}^T \mathbf{a}_j.$$

Unfortunately, the opposite of Proposition 9.7.2 is not true. Namely, if $T_j \leq 0$ then anything can happen, including the favorable case of a nondegenerate step. The positive side of this negative result is that we can give preference to those candidates for which $T_j \leq 0$ as long as there are such variables to exclude the definitely poor choices characterized by (9.113).

The extra work of computing T_j according to (9.113) is the same as computing \hat{V}_j by (9.111).

It is important to stress that the use of a column selection method based on the heuristics of Proposition 9.7.2 can only be recommended if the basis is degenerate. It is never harmful as it will always selects real improving candidates with respect to w thus satisfying an important principle of the simplex method. However, if the basis is nondegenerate it should not be used as the quality of selection can be far from ideal, at least from computational point of view. The ‘result’ can be too many small iterations.

The other heuristics based on the approximation of Proposition 9.7.1 is also worth trying though currently there is not enough computational evidence about its effectiveness.

9.7.2 Wolfe’s ‘ad hoc’ method

Wolfe in [Wolfe, 1963] describes a procedure, which he calls ‘ad hoc’ method, to cope with degeneracy in a theoretically sound and computationally efficient way. It proved to be particularly useful in case of solving the LP relaxation of some combinatorial problems (Ryan and Osborne [Ryan and Osborne, 1988]) It can be characterized as a kind of ‘virtual perturbation’.

The ‘ad hoc’ method is applicable in phase-2 where preserving feasibility is an essential feature of the simplex method. We present Wolfe’s procedure in a slightly modified form. The main difference is the special treatment of the type-0 basic variables that are at zero level. Namely, we give priority to the removal of type-0 variables from the basis as they may cause (or contribute to) stalling in phase-2. For simplicity, the method is described in a theoretical framework, i.e., without explicit inclusion of feasibility tolerances. Its extension to a computationally suitable form is rather straightforward. The problem in question is CF-1 of LP, see (1.20).

The degeneracy sets of the basic variables are defined slightly differently from (9.107) and (9.108):

$$\mathcal{D}^+ = \{i : \beta_i = 0, \text{ type}(\beta_i) \in \{1, 2\}\}, \quad (9.114)$$

$$\mathcal{D}^- = \{i : \beta_i = v_i, \text{ type}(\beta_i) = 1\}, \quad (9.115)$$

$$\tilde{\mathcal{D}} = \{i : \beta_i = 0, \text{ type}(\beta_i) = 0\}. \quad (9.116)$$

Type-0 variables are listed separately as they get a special treatment. In phase-2 they are always equal to zero.

If x_q is to enter the basis positively it will result in a nondegenerate iteration if $\alpha_q^i \leq 0$ for all $i \in \mathcal{D}^+$, $\alpha_q^i \geq 0$ for all $i \in \mathcal{D}^-$ and $\alpha_q^i = 0$ for all $i \in \tilde{\mathcal{D}}$. It is just a different way of writing (9.112).

It is known that the solution is unbounded (x_q is not blocked) if $\text{type}(x_q) \in \{2, 3\}$ and

$$\alpha_q^i \leq 0, \forall i \text{ such that } \text{type}(\beta_i) \in \{0, 1, 2\}, \quad (9.117)$$

$$\alpha_q^i \geq 0, \forall i \text{ such that } \text{type}(\beta_i) \in \{0, 1\}. \quad (9.118)$$

A direction for which this holds is called a *direction of recession*. If such a direction, restricted to the degenerate positions, can be found a nondegenerate iteration can be made. This remarks sets the scene for the forthcoming algorithm. However, there are some preliminaries that are important from computational point of view.

At the first iteration when a $\theta = 0$ (zero steplength) is encountered we do the following. If $\tilde{\mathcal{D}}$ is nonempty choose the outgoing from among the positions in $\tilde{\mathcal{D}}$ with the largest pivot element, i.e., $p : |\alpha_q^p| \geq |\alpha_q^i|, i \in \tilde{\mathcal{D}}$. In this way priority is given to the removal of type-0 variables from the basis which usually leads to the reduction of the detrimental effects of degeneracy. If $\tilde{\mathcal{D}}$ is empty we enter Wolfe's algorithm.

Let $\mathcal{D} = \mathcal{D}^+ \cup \mathcal{D}^-$. Now a direction of recession for \mathcal{D} is sought. The algorithm uses the notion of *depth of degeneracy*. At the outset every position in \mathcal{D} has a zero depth. Every time a position with the largest depth undergoes a ‘virtual perturbation’ the degeneracy deepens. On the other hand, when a direction of recession is found for the subproblem consisting of the deepest positions, the degeneracy shallows. The only thing that needs to be noted of a position in \mathcal{D} is its depth which will be denoted by $g_i, i \in \mathcal{D}$.

Initialization: Set $g_i = 0, i \in \mathcal{D}$. Assuming that an improving candidate x_q has been selected and α_q is available, one iteration of the algorithm can be described by the following steps.

Step 1. For each i with $\beta_i = 0$ replace β_i by a small positive random number δ_i . Similarly, for each i that have $\beta_i = v_i$ replace β_i by

$v_i - \delta_i$. Note, δ_i is randomly chosen for each position and should be such that the values after the perturbation remain feasible. In both cases replace g_i by $g_i + 1$ to signify that the degeneracy has deepened.

Step 2. Let $G = \max_i\{g_i\}$. Perform a ratio test with positions for which $g_i = G$. If a pivot row p is found with $\theta = \beta_p/\alpha_q^i$ or $\theta = (\beta_p - v_p)/\alpha_q^p$ go to Step 3, otherwise go to Step 4.

Step 3. Apply a special update. The outgoing variable leaves the basis at its exact bound. The basic variables in \mathcal{D} are updated such that for any position with $g_i < G$ everything remains unchanged. For positions with $g_i = G$ replace β_i by $\beta_i - \theta\alpha_q^i$ for $i \neq p$ and β_p by $x_q + \theta$. Set $g_p = G$.

Basic variables outside \mathcal{D} remain unchanged as the current pivot step corresponds to a degenerate iteration for the original problem.

Choose a new incoming column and return to Step 1. If none can be chosen the current basis is optimal. Solution needs to be recomputed with restored bounds for variables in \mathcal{D} .

Step 4. As a pivot row was not found we have a direction of recession for the subproblem consisting of rows with $g_i = G$, consequently the degeneracy has shallowed.

If $G > 0$, for each i with $g_i = G$ replace β_i by zero or upper bound (as appropriate) and g_i by $g_i - 1$. Return to Step 2.

Otherwise, if $G = 0$ the problem is unbounded.

From the above description it can be seen that degeneracy can deepen if the minimum ratio is not unique in the ratio test. However, as the iterations are always nondegenerate, the number of elements in the deeper level of degeneracy will be at least one less. It can go down no deeper than m where just one element can be present. The purpose of the random choice of the perturbation is to reduce the chance of deepening.

An important advantage of the method is that it does not introduce infeasibilities. Therefore, when a real nondegenerate step is made the solution remains feasible. Further advantages are the moderate increase in the computational effort and the simplicity of implementation.

There is a potential conflict between the requirement of flexibility in choosing the pivot element and the uniqueness ensured by the method. It can be overcome by looking for a good pivot first and performing the perturbation afterwards to let the chosen position have the smallest ratio. Among other implementation issues it is worth mentioning that the recomputation of G in Step 2 can be replaced by an update. Furthermore, the degenerate positions with identical depth can be kept on

easily maintainable linked lists. The total length of the lists is never more than m as a basic position can be on at most one list.

If we use some anti-degeneracy pricing strategy it usually will ‘play into the hands’ of the above procedure. If the idea of proposition 9.7.2 is used then degeneracy sets \mathcal{D}^+ and \mathcal{D}^- are restricted to positions with $g_i = G$ which results in a speed-up of the necessary BTRAN operation.

As a final remark, we note that this is a ‘degeneracy aware’ method. It is activated if degeneracy prevents a change in the value of the objective function. All degenerate positions have to be identified which requires some extra work. It is important how the near degenerate positions are identified and handled (included or not in the degeneracy sets depending on the choice of ε_d in (9.107) and (9.108)). Because of the setup overhead of the method it is advisable to delay its activation until some specified, say `maxcycle` number of consecutive degenerate steps have been made. It gives a chance that short rallies of nonimproving steps are treated by the standard simplex mechanism. The value of `maxcycle` can be the number of iterations between two refactorizations.

The ‘ad hoc’ method performs very well in practice, in particular in case of LP relaxation of combinatorial optimization problems where degeneracy is clearly identifiable and near degenerate variables are less frequent. The depth of degeneracy hardly ever goes beyond 2 if the perturbations are properly selected. Therefore, the amount of extra computations remains very moderate.

9.7.3 Expanding tolerance

Stallingstalling can be prevented if at least a minimum progress is made in every iteration.

If degeneracy is present a zero steplength may be determined by the ratio test. If such a pivot step is made there is no change in the objective value. This situation carries the theoretical danger of cycling. In practice, degeneracy can lead to long nonimproving sequences which was already referred to as stalling. Gill et al. [Gill et al., 1989] proposed a method that ensures at least a minimum progress in every iteration, if necessary, at the expense of slightly violating the feasibility of some (or all) basic variables. This ensures that no basis is repeated. The emerging infeasibility can be kept under control and eliminated when a relaxed optimal solution is reached. In this section we present the details of their method, called EXPAND (*EXPanding-tolerance ANti-Degeneracy*) procedure as it deserves considerable attention for several reasons.

EXPAND shows similarities with Harris's ratio test and Wolfe's 'ad hoc' method but differs from them in some important aspects.

EXPAND assumes that an incoming variable has been chosen and its aim is to determine the outgoing variable in such a way that at least a guaranteed minimum positive steplength is achieved. It uses a Harris-type two pass scheme for finding a 'maximum pivot' for stability reasons.

In the description we use the LP problem in CF-1. In EXPAND the bound constraints are relaxed to

$$-\delta_f \mathbf{e} \leq \mathbf{x} \leq \mathbf{u} + \delta_f \mathbf{e},$$

where \mathbf{e} is an n -dimensional vector with all components equal to 1 and δ_f is the 'master' feasibility tolerance. Any violation of the bounds by more than this tolerance is treated as real infeasibility. The procedure uses some other tolerances that are always smaller than δ_f . There is a 'working' feasibility tolerance, δ_k , which is slightly increased before every iteration. The increment is denoted by τ .

An iteration starts with the selection of an improving variable x_q and computing its updated form $\alpha_q = \mathbf{B}^{-1}\mathbf{a}_q$. After that the current value of the working feasibility tolerance is determined from its old value of δ_{k-1} as $\delta_k = \delta_{k-1} + \tau$.

The first pass of the Harris ratio test is performed with tolerance δ_k in place of ε_f in (9.39). It tests the \hat{t}_i ratios and provides the index p of the pivot row and the minimum ratio θ_1 . It is to be noted that the original version of EXPAND does not give type-0 variables the special treatment we have shown in the modified Harris ratio test. Pass 1 is followed by Pass 2, again, in the way described there. Therefore, for the $t \geq 0$ case, formally, we have the following steps

1 Pass 1.

$$\hat{t}_i = \begin{cases} \frac{\beta_i + \delta_k}{\alpha_q^i}, & \text{if } \alpha_q^i > \varepsilon_p, \\ \frac{\beta_i - v_i - \delta_k}{\alpha_q^i}, & \text{if } \alpha_q^i < -\varepsilon_p. \end{cases}$$

Let $\theta_1 = \min\{\hat{t}_i\}$. Assume the incoming variable is not blocked by its own upper bound, i.e., $\theta_1 \leq u_q$.

2 Pass 2.

Compute the ratios again, this time with the exact bounds

$$t_i = \begin{cases} \frac{\beta_i}{\alpha_q^i}, & \text{if } \alpha_q^i > \varepsilon_p, \\ \frac{\beta_i - v_i}{\alpha_q^i}, & \text{if } \alpha_q^i < -\varepsilon_p. \end{cases}$$

Put the row indices of ratios smaller than θ_1 into \mathcal{T} , i.e.,

$$\mathcal{T} = \{i : t_i \leq \theta_1\}.$$

- 3 Choose $p \in \mathcal{T}$ such that $|\alpha_q^p| \geq |\alpha_q^i| \forall i \in \mathcal{T}$ defining $\hat{\theta} = t_p$. Set $\theta = \max\{\hat{\theta}, 0\}$.

So far it is the unmodified Harris ratio test. It will give a zero step if $\hat{\theta} \leq 0$. At this point EXPAND takes over as it wants to ensure that a minimum acceptable steplength of $\theta_{\min} = \tau/\alpha_q^p > 0$ is always achieved. Therefore, the EXPAND steplength θ_E is determined as

$$\theta_E = \max\{\theta, \theta_{\min}\}. \quad (9.119)$$

The consequences of this choice on the outgoing variable are the following.

- 1 If the leaving variable $\beta_p \equiv x_{k_p}$ was feasible with a distance from its nearest bound more than τ then this is a nondegenerate step and x_{k_p} leaves the basis at its exact bound.
- 2 If x_{k_p} was closer to its bound than τ then a forced minimum step is made and its bound will be violated by at most τ .
- 3 If x_{k_p} was already infeasible within the working tolerance then it will become more infeasible by τ .

The latter two cases correspond to a degenerate iteration. The infeasibility of the outgoing variable will deteriorate by at most τ to less than $\delta_k + \tau$. However, $\delta_k + \tau$ is nothing but δ_{k+1} in the next iteration. Therefore, the variable will remain feasible thanks to the expanding feasibility tolerance. The only important thing is to record the actual value of this new nonbasic variable (which is different from its bound). As a result of the update some basic variables may become infeasible but they and also the already infeasible ones remain within $\delta_k + \tau$ of their violated bounds.

The EXPAND iteration will result in a definite progress in the objective function. Because this improvement is strictly monotone, no basis can be repeated, thus the method cannot cycle in this way.

It is to be noted that EXPAND can be used also in phase-1 of the simplex method.

It is undesirable to let the tolerance expand too much as it would result in the solution of a rather perturbed problem or would lead to divergence. Therefore, the expanding tolerance must be kept under control.

The number of times the feasibility tolerance is allowed to expand is limited to K which is a parameter of the procedure. As the master tolerance δ_f is never exceeded, $\delta_k \leq \delta_f$ for $k \leq K$.

Typical recommended values for the parameters, if double precision arithmetic is used with a relative accuracy of 10^{-16} :

δ_f	$= 10^{-6}$	master feasibility tolerance,
K	$= 10,000$	maximum number of simplex iterations without resetting,
δ_0	$= 0.5\delta_f$	initial feasibility tolerance,
δ_K	$= 0.99\delta_f$	maximum feasibility tolerance during an expanding sequence,
τ	$= (\delta_K - \delta_0)/K$	increment of the feasibility tolerance.

To enable more than K iterations, a *resetting* takes place after every K expanding steps. Resetting is also necessary when an optimal solution with the expanded bounds has been found to determine the solution in terms of the exact bounds of the nonbasic variables.

Resetting assigns the starting value δ_0 to the working tolerance and also sets $k = 0$. Additionally, every nonbasic variable that lies within δ_f of one of its bounds is moved to the exact bound. The number of such moves is counted if the distance travelled exceeds some other tolerance which is in the magnitude of τ . If the count is positive the basic variables are recomputed with the current values of the nonbasics to satisfy $\mathbf{Ax} = \mathbf{b}$ to machine precision. After that, simplex iterations resume with incrementing k by one and setting $\delta_1 = \delta_0 + \tau$.

As a result of resetting the basic solution may become infeasible which triggers phase-1 iterations to restore feasibility. Experience shows that in case of well behaved problems feasibility is generally preserved. In the few cases when feasibility is lost the recovery is generally quick and the objective value does not deteriorate noticeably. With less well behaved problems, however, all sorts of things can happen, like long recovery, serious loss in the objective or even feasibility may not be regained before the next reset. There are a couple of possibilities to try to resolve such problems. First of all, \mathbf{A} has to be scaled properly. Next, K can be increased to a much larger value (with the automatic adjustment of the other parameters, see table above). Furthermore, the master feasibility tolerance can be changed, most likely increased from 10^{-6} to, say, 10^{-5} .

If feasibility is lost when a resetting is made at an optimal solution then some extra caution is required. Though, with well behaved problems the solution is likely to remain feasible, and, therefore optimal, sometimes a few iterations may be needed. The best is to use the dual simplex (discussed in detail in chapter 10) as dual feasibility is not affected. However, if primal is used with EXPAND for the ‘reoptimization’ then at the new optimal solution one more resetting has to be done. Unfortunately, it can lead to infeasibilities again though probably to a lesser

extent. To avoid this sort of cycling it is reasonable to limit the number of resets to a small value (like 2, or at most 3) if they were triggered by an optimal solution.

EXPAND has several nice features. It is a ‘degeneracy unaware’ method which means the overhead of identifying degeneracy and the degenerate positions is automatically avoided. There is no logical overhead in the ratio test. It also supports Harris’s idea of ‘maximal pivot’ for numerical stability.

It is very simple to implement EXPAND. The extra storage requirement of keeping the values of the nonbasic variables not at their bound is easy to accommodate. In modern implementations the entire solution vector \mathbf{x} is stored anyhow (for many different reasons). It is important to remember that after refactorization (reinversion) of the basis, when the solution is recomputed, the $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{R}\mathbf{x}_{\mathcal{R}}$ form should be used to account for variables in \mathcal{R} which are at upper bound or have a value different from one of their bounds.

Since EXPAND performs very well on small to medium size problems it is a good idea to include it in a solver. However, it has to be remembered that in case of some ill behaved and large scale LP problem instances resetting may lead to difficulties outlined above.

9.7.4 Perturbation, shifting

The occasional failure of the EXPAND procedure can be attributed to the unnecessary expansion of the bounds for all variables not only the ones involved in degeneracy. This may lead to long recovery of feasibility or even to failing to achieve it if the primal algorithm is used for this purpose. In some cases the flexibility of ‘pivoting for size’ may also suffer increasing the chances of numerical difficulties and the magnitude of the real infeasibility may be in the order of the tolerances in phase-1. While these undesirable events are quite rare with EXPAND, a robust solver should be able to handle them properly.

Interestingly enough, some quite obvious heuristics can be extremely effective in coping with degeneracy. While the ideas of perturbation and shifting lack some theoretical beauty their practical usefulness is beyond any doubt.

In case of perturbation the values of the basic variables are changed while in shifting we change the individual bounds of some of the variables. Shifting was originally called *bound shifting*. The two approaches can be viewed as practically more or less equivalent. The real differences lie in the implementation.

9.7.4.1 Perturbation

Perturbation has already been discussed within the framework of Wolfe's 'ad hoc' method to cope with degeneracy. In fact, perturbation can be used on its own without that framework. It is a heuristics which proved to be very effective if implemented with some care. Such a procedure can be outlined as follows.

- 1 If no progress is made in the objective function for `maxcycle` iterations identify the \mathcal{D} index set of degenerate positions. Do not include type-0 basic variables in \mathcal{D} to be able to remove them from the basis as quickly as possible.
- 2 Apply random perturbation on variables in \mathcal{D} so that they remain feasible.
- 3 Apply the simplex method. If a position in \mathcal{D} is used as pivot perform a solution update on variables in \mathcal{D} only. If a pivot outside \mathcal{D} is chosen then a nondegenerate step is made. In this case, restore the degenerate values in \mathcal{D} before update and set $\mathcal{D} = \emptyset$.

There can be several refinements and tactical variations to the above scheme.

9.7.4.2 Modified perturbation of MPSX

Benichou et al. [Benichou et al., 1977] describe a perturbation idea that deserves attention. They have implemented it in IBM's MPSX package. Here we present it in a slightly modified form to include the random choice of perturbation which was not there in its original form. For clarity of the description, it is now assumed that there are no upper bounded variables in the problem. Their inclusion can be implemented by some obvious amendments.

\mathbf{A} is partitioned in the usual way $[\mathbf{B} \mid \mathbf{R}]$, where \mathbf{B} is the basis and \mathbf{R} is the remaining part of \mathbf{A} at a certain iteration. The solution vector is partitioned accordingly.

At an iteration when it becomes necessary to deal with degeneracy, the solution in full form is

$$\mathbf{Bx}_B + \mathbf{Rx}_R = \mathbf{b}$$

from which the actual basic solution can be expressed. In the present context it is denoted by $\hat{\mathbf{b}}$ to mark that this was the solution where degeneracy started receiving a special treatment

$$\mathbf{x}_B + \mathbf{B}^{-1}\mathbf{Rx}_R = \mathbf{B}^{-1}\mathbf{b} = \hat{\mathbf{b}}.$$

Assume ξ_i is a random numbers from a given interval, say, $[10^{-3}, 10^{-2}]$. Define a perturbation vector ϵ by

$$\epsilon_i = \begin{cases} \xi_i & \text{if } |\hat{b}_i| \leq \varepsilon_f \text{ and type}(\beta_i) \neq 0, \\ 0 & \text{otherwise,} \end{cases}$$

and add it to $\hat{\mathbf{b}}$ giving

$$\mathbf{x}_{\mathcal{B}} + \mathbf{B}^{-1}\mathbf{R}\mathbf{x}_{\mathcal{R}} = \hat{\mathbf{b}} + \mathbf{I}\epsilon. \quad (9.120)$$

Obviously, no infeasibilities are generated by this perturbation. However, the original problem changes to

$$\mathbf{B}\mathbf{x}_{\mathcal{B}} + \mathbf{R}\mathbf{x}_{\mathcal{R}} = \mathbf{b} + \mathbf{B}\epsilon. \quad (9.121)$$

(9.121) is obtained by multiplying both sides of (9.120) by \mathbf{B} . From this point on the perturbed problem (with right-hand side as given in (9.121)) is used and solved to optimality. If degeneracy occurs again, a similar action can be taken. The random perturbation makes it very unlikely that such a move will be needed.

When optimality is reached with the perturbed problem the original right-hand side is restored. As a result, the recomputed basic solution may be infeasible. However, the basis is dual feasible and the dual algorithm can be used to obtain primal feasibility, thus optimality for the original problem.

9.7.4.3 Shifting

The starting idea of shifting is the observation that in case of the Harris ratio test a negative step can be defined which, if implemented, would deteriorate the objective value. It raises the danger of ‘numerical cycling’. In such a case Harris sets the steplength to zero which, on the other hand, slightly alters the LP problem that can lead to problems with feasibility later on. One way of resolving this problem is to use the EXPAND procedure described in section 9.7.3. Another possibility is to apply bound shifting. In this case we need the explicit use of lower bounds whether they are zero or not. The lower bound of basic variable i is denoted by λ_i .

If `maxcycle` number of nonimproving iterations have been made then change (perturb) the bound of those basic variables which violate their bound by at least ε_f . Such small infeasible values are the result of the Harris ratio test. As we assume that all type-0 variables retain their zero value in phase-2 they are not included in shifting. If $-\varepsilon_f \leq \beta_i < \lambda_i$ then we change the lower bound to a randomly chosen smaller value so

that β_i is well away from it. A possible choice is $\lambda_i := \lambda_i - \epsilon_i$ where ϵ_i is a random number in the $[100\epsilon_f, 1000\epsilon_f]$ interval. This gives space for a movement by several orders of magnitude larger than the feasibility tolerance. If $v_i < \beta_i \leq v_i + \epsilon_f$ then the upper bound is changed to $v_i + \epsilon_i$, where ϵ_i is also a random number from the same interval. The consequence of these bound shifts is that a positive feasible step can be made and the outgoing variable leaves at its newly defined exact bound. The bounds are retained in the subsequent iterations but need to be removed as soon as possible. The latest is when an optimal solution is achieved with the new bounds. The restoration of the original bounds will very likely lead to infeasibilities but dual feasibility is maintained. Therefore, the dual algorithm can be used to regain primal feasibility, thus optimality.

Bound shifting can be applied also in phase-1.

In practice, bound shifting performs remarkably well. One of its attractive points is that it does not affect the pricing strategy used. Actually, nowadays it is the method of choice for solving degenerate large scale problems, even if there is no strong theoretical backing for it but the computational evidence is overwhelming.

Regarding its implementation, the explicit lower bound version of the simplex algorithm can be recommended.

9.8. Starting bases

To facilitate the simplex method an initial basis is needed. As in both computational forms (1.20) and (1.26) the constraint matrix is of full row rank there exist at least one set of m linearly independent column vectors that form a basis. There are several possible ways of choosing such a set. It can be expected that if more effort is made then a better starting basis can be found, ideally an optimal basis (if one exists). By ‘better’ we mean that a solution to the problem is obtained by less overall computational effort including the creation of the starting basis itself. Therefore, the requirement of any starting procedure is to create a good starting basis with reasonable computational effort.

In this section we discuss methods for selecting a starting basis. We will concentrate on the practical usefulness of the different approaches. Some issues of implementation will also be discussed.

9.8.1 Logical basis

In computational forms (1.15) and (1.24) of the LP problem introduced in Chapter 1 every row has a logical variable associated with it.

Their column vectors form a unit matrix that can be chosen as an initial basis for the simplex method, i.e., $\mathbf{B} = \mathbf{I}$.

While the logical basis is extremely simple it cannot be completely ignored. It has at least three distinct advantages over any other initial bases: (a) its creation is instantaneous, (b) the corresponding \mathbf{B}^{-1} is available without any computations being \mathbf{I} itself and (c) the first simplex iterations are very fast as the operations with such an inverse are fast and the length of the representation of \mathbf{B}^{-1} (or the LU factorization) grows just slowly.

Depending on how the nonbasic variables are fixed we can talk about *lower logical*, *upper logical* and *mixed logical* bases. A logical basis is sometimes also referred to as *trivial basis*. In case of the lower logical basis all nonbasic (in this case the structural) variables are set to their zero lower bound. Thus the basic solution is $\mathbf{x}_B = \mathbf{b}$.

If there are type-1 structural variables in the problem then it may be assumed that their upper bounds were given in the belief that they will be active. Therefore, in this case we may try to set all nonbasic type-1 structural variables to their upper bound resulting in the following basic solution

$$\mathbf{x}_B = \mathbf{b} - \sum_{j \in \mathcal{U}} a_j^i u_j,$$

where \mathcal{U} denotes the index set of type-1 structural variables.

A further variation can be obtained if only those type-1 structurals are set to their upper bound that have a favorable objective coefficient, i.e., $c_j < 0$, in case of a minimization problem and $c_j > 0$ if the problem is a maximization. Such a basis can be referred to as mixed logical basis. In real-world problems the upper logical and mixed logical bases are quite often the same.

When creating a logical basis the feasibility of the corresponding solution is ignored that usually (but not always) results in a long rally in phase-1.

It is not uncommon that many or all of the constraints are modeled such that the right-hand side coefficients are zero. In the latter case the lower logical basis gives a completely degenerate but feasible solution. Depending on how degeneracy is handled in the solution algorithm it can be a good or a bad starting basis.

Despite the advantages of a logical basis it is not used exclusively. The reason is that, in general, it leads to substantially larger iteration counts than other starting bases, though there are counter examples in favor of the logical basis.

9.8.2 Crash bases

Experience has shown that a starting basis with many structural variables in it generally leads to a better initial solution than a logical basis. Of particular importance are the triangular bases that have some unbeatable features, like (a) their inverse can be represented by exactly the same number of nonzero entries as the basis itself (no fill-in), (b) they are numerically very accurate, (c) their creation is usually very fast, and (d) operations with them are fast. Therefore, it is quite a natural idea to develop an algorithm that creates a largest possible triangular basis matrix the columns of which are the structural columns of A or unit vectors of logical variables.

The identification whether a given basis is triangular requires logical operations only. The same is true for the selection of a (not necessarily square) triangular matrix from a set of vectors.

Creating a triangular basis can be conceived as starting up with the identity matrix I and replacing its columns by columns of structural variables in such a way that the new matrix remains triangular.

Two types of triangular bases B can be defined in the LP context. In the first, the *lower triangular*, case the nonzeros of B are located in and below the main diagonal. In the second, the *upper triangular*, case the nonzeros of B are located in and above the main diagonal. In both cases the main diagonal is zero-free.

In this section the structural and logical part of the matrix of constraints will be distinguished by the partition $\mathbf{A} = [\hat{\mathbf{A}} \mid \mathbf{I}]$.

The simplest way to trace a possible triangular structure is to use row and column counts, R_i and C_j , which are defined as the number of nonzeros in row i of $\hat{\mathbf{A}}$, and the number of nonzeros in column j of $\hat{\mathbf{A}}$, respectively.

The diagonal elements of a lower triangular basis have only zero elements on their right. Therefore, for the first diagonal element (upper left corner) one can select a pivot row i such that $R_i = \min_k\{R_k\}$. The pivot column in this row is uniquely determined if $R_i = 1$. However, if $R_i > 1$, there are alternative choices. If we want to keep the nonzero count in B low then a column with nonzero in row i having the smallest column count has to be chosen. The selection is usually based on some *additional criteria* if there is a tie. The selected element becomes the current pivot and the pivot position is logically permuted (by row and column permutations) to the top left corner. All columns having a nonzero element in row i are marked unavailable for further consideration. This ensures that the remaining columns do not have to be transformed at inversion/factorization. Row and column counts are

recomputed (updated) for the remaining rows and columns (the *active part*) of $\hat{\mathbf{A}}$. These steps are repeated for the active part of $\hat{\mathbf{A}}$. This simple procedure is the basis of the lower triangular crash procedures. The pivot order is the natural order as the pivot positions are identified.

Any later selected column in this procedure is such that it has a zero element in all the previous pivot positions. This may impose a limit on the success of finding further triangular components. In any case, when the procedure terminates we have vectors that can be permuted into lower triangular form. They may not make a full basis but the unused pivot positions remain filled by the corresponding unit vectors resulting in a full lower triangular basis \mathbf{B} .

Finding an upper triangular (sub) matrix can be done in the following way. Select a column with minimum C_j . If this is equal to 1 then the pivot row is uniquely determined and this position is permuted to the top left position, the pivot row and column are marked unavailable for further consideration, column counts are updated, search for new $C_j = \min_k \{C_k\}$ continues. If $C_j > 1$ then we have to determine which position in column j will be pivotal. This is usually done on the basis of minimum row count for the rows with $a_j^i \neq 0$. Again, some additional criteria may be applied in case of a tie. Having selected a pivot position in the given column we mark this column unavailable; all rows that have nonzeros in this column are also marked unavailable. Column and row counts are updated for the remaining part of $\hat{\mathbf{A}}$ and the procedure is repeated.

Later columns in this procedure are characterized by having a nonzero element in at least one of the rows that are still available for pivoting. If we do not get a full upper triangular basis then the unused pivot positions remain filled by the corresponding unit vectors of \mathbf{A} which leads to a complete lower triangular basis \mathbf{B} . The pivot order in the upper triangular case is the reverse, that is, the pivot position found last is used first.

In each step of the lower triangular procedure we mark unavailable one row (the pivot row) and all the columns that intersect with this row. In the upper triangular procedure one column (the pivot column) becomes unavailable and all rows that intersect with this column. The two procedures may find triangular parts of different size. The quality of the basis is determined by several factors, such as density, condition number, feasibility, and the corresponding objective value. These features or their combination play an important role in breaking row or column ties during the selection.

The above description of triangular crash procedures provides a conceptual framework and the actual procedures are based on these ideas but involve a number of other considerations.

If a procedure does not take into account the actual numerical values of the coefficients but only utilizes the location of nonzeros, we say it is a *symbolic crash* procedure. If the numerical values of the coefficients are taken into account then this is defined as a *numerical crash* procedure.

9.8.2.1 CRASH(LTSF): A ‘most feasible’ triangular basis

In this section we describe a *Lower Triangular Symbolic* crash procedure designed for *Feasibility* and we call it CRASH(LTSF).

We proceed with the general procedure of the lower triangular symbolic crash as described above. If at every stage the selection is unique (i.e., there is a unique minimum row count and a unique minimum column count for the positions in the selected row) then this procedure is just a lower triangular crash CRASH(LTS). However, in practice, this happens very rarely and in general there are multiple choices at each step. The main idea of CRASH(LTSF) is that if column selection for triangularity is not unique then we give preference to

- removal of a logical variable with small feasibility range, and
- inclusion of a structural variable with large feasibility range.

It is expected that this will lead to a ‘more feasible basis’.

The feasibility ranges are ranked in accordance with the usual type specification of the variables. The definition of row and column priorities is given in Tables 9.1 and 9.2. In these tables, “Row-type” refers to the type of the logical variable of a row and, similarly, “Col-type” refers to the type of the corresponding structural variable.

Table 9.1. Row priority (RP) table

Row-type	Priority	Comments
0	3	Highest priority is equality row
1	2	Range type row
2	1	“ \leq ” or “ \geq ” type row
3	0	Lowest priority is free row

In some sense the most favorable combination is to *replace a Type-0 logical by a Type-3 structural*.

Table 9.2. Column priority (CP) table

Col-type	Priority	Comments
0	0	Lowest column priority is fixed variable
1	1	Type-1 (bounded) variable
2	2	Type-2 (non-negative) variable
3	3	Highest column-priority is free variable

The row and column selections will be based on a row and a column priority function $RPF(i)$ and $CPF(j)$, respectively. In order to formally define them the following notations need to be introduced:

- $RT(i)$ = Type of the logical variable of row i .
- $RC(i)$ = Number of nonzeros in row i of the active columns of \bar{A} .
- $RI(i)$ = Index set of active columns intersecting row i .
- $CT(j)$ = Type of column j .
- $CC(j)$ = Number of nonzeros in column j of the active rows of \bar{A} .
- AR = Index set of available (active) rows.

The *Row Priority Function* is defined as:

$$RPF(i) = RP(RT(i)) - 10 \times RC(i).$$

Here, 10 is just a magic number that helps separate the values of $RPF(i)$ well. The row selection is made by finding a row with maximum value of this function:

$$r = \max_{i \in AR} \{RPF(i)\}$$

Similarly, the *Column Priority Function* is defined as:

$$CPF(j) = CP(CT(j)) - 10 \times CC(j).$$

Having selected row r we trace this row for nonzero entries and if such an entry is found we evaluate $CPF(j)$ for that column. Finally, the selected column is determined by the index k , such that

$$k = \max_{j \in RI(r)} \{CPF(j)\}$$

Of course, there is a possibility that ties are still present. However, now this means that the candidates are equally good from the point of

view of feasibility ranges. In the present version of the algorithm we simply take the first position in the tie.

The steps of the CRASH(LTSF) algorithm are summarized in Algorithm 3.

Algorithm 3 CRASH(LTSF) to create a lower triangular starting basis

Require: Matrix $\mathbf{A} = [\hat{\mathbf{A}} \mid \mathbf{I}]$.

Ensure: A lower triangular basis \mathbf{B} to \mathbf{A} .

- 1: **Step 1:** Set up priority tables 9.1 and 9.2.
 - 2: **Step 2:** Mark fixed columns and free rows unavailable.
 - 3: **Step 3:** Initialize basis to all-logical, $\mathbf{B} = \mathbf{I}$. The rest of \mathbf{A} (which is $\hat{\mathbf{A}}$) is called *active*.
 - 4: **Step 4:** Compute $RC(i)$ and $CC(j)$ for the active part of $\hat{\mathbf{A}}$.
 - 5: **Step 5:** Make row selection by *RPF*.
 - 6: **Step 6:**
 - 7: if Row selection successful then
 - 8: (a) Make column selection by *CPF*.
 - 9: (b) Redefine the active part of $\hat{\mathbf{A}}$ and update the corresponding row and column counts.
 - 10: (c) Go back to row selection (Step 5).
 - 11: end if
 - 12: **Step 7:** Terminate CRASH(LTSF).
-

For the efficient implementation of CRASH(LTSF) a rowwise data structure of $\hat{\mathbf{A}}$ is needed. Additionally, using a doubly linked list of rows with equal row counts makes the implementation of this procedure work very fast. Such a procedure can easily be created with the help of the tools and procedures previously described in sections 5.4 and 5.5.

As a result, time spent in CRASH(LTSF) is negligible compared to the total solution time. For example, a 95% triangular part of a problem with 20,000 rows can be found in less than 0.05 seconds on a 1GH Pentium III processor.

We also note that the above algorithm can be refined in several different ways. Their full elaboration is not given here. We just indicate that taking into account the sign of the potential pivot elements can further improve the chances of obtaining a ‘more feasible’ basis. The other important point can be the consideration of the sign of the objective coefficients of the columns in the tie to obtain a better basis in terms of the objective function. Furthermore, preference can be given to positions with larger pivot elements or a relative pivot tolerance can be applied. The latter means that any element in a column is treated as

zero that does not satisfy

$$|a_j^i| \geq u \max_k \{|a_j^k|\},$$

where u is a parameter with typical values $0.01 \leq u \leq 0.1$. This treatment of potential pivots is sometimes beneficial even if in this procedure every pivot element is ‘original’ and not a computed value, therefore free from computational error.

CRASH(LTSF), even in the presented simple form, is a good performer. It can find a remarkably large lower triangular basis \mathbf{B} . It is definitely a candidate for inclusion in a modern implementation of the simplex method.

9.8.2.2 CRASH(ADG) for a ‘least degenerate’ basis

In many real life problems the right-hand side vector \mathbf{b} may contain many zero components. As a result of a numeric crash procedure, some of them may change to nonzeros but still there is a good chance that an advanced basis and the corresponding solution will be primal degenerate. The following crash algorithm maintains triangularity and attempts to pivot on rows i for which $b_i \neq 0$. Every update of the right-hand side with such a pivot can increase the number of nonzero values in β (the updated \mathbf{b}) thus reducing the degree of degeneracy.

Without loss of generality, we assume that $\beta = \mathbf{B}^{-1}\mathbf{b}$ which is equal to \mathbf{b} at the beginning for the all-logical basis. We define set \mathcal{H}_1 to contain the row indices for which the corresponding $\beta_i \neq 0$:

$$\mathcal{H}_1 = \{i : \beta_i \neq 0, k_i \in \mathcal{I}\} \quad (9.122)$$

where k_i denotes the original index of the i -th basic variable and \mathcal{I} is the index set of the logical variables in \mathbf{A} . Here, $k_i \in \mathcal{I}$ signifies that row i was not pivotal yet.

An ‘anti-degeneracy crash’ procedure, called CRASH(ADG), can be described verbally by the following 3-step algorithm:

- 1 Scan the available structural columns of \mathbf{A} and compute the following *merit number* for each of them:

$$j_k = \sum_{\substack{i \in \mathcal{H}_1 \\ a_k^i \neq 0}} 1$$

If the summation set is empty then the sum is defined to be zero. Verbally, j_k is the number of nonzeros in column k that intersect the set of nonzero positions \mathcal{H}_1 of β .

If there is no eligible column or $\max_k j_k = 0$ then the algorithm terminates. Otherwise, column q is selected for which $j_q = \max_k j_k$. This criterion prefers columns which have more nonzeros in potential pivotal columns. If this selection is not unique then we can break the tie by selecting one of them with the largest overall nonzero count to ensure a larger change in the nonzero structure of β .

- 2 Whenever $j_q > 1$ we have some freedom to actually determine the pivot row. For column q we define the index set of all possible pivots:

$$\mathcal{H}_2 = \{i : i \in \mathcal{H}_1, a_q^i \neq 0\}$$

and compute the following quantities:

$$t_i = \beta_i / a_q^i, \text{ for } i \in \mathcal{H}_2 \quad (9.123)$$

The t_i -s determine different nonzero values of the incoming variable assuming the outgoing variable leaves at 0 level. (The upper bounded version can be worked out easily.) Since all $i \in \mathcal{H}_2$ positions are such that they determine a nondegenerate basis change, if $j_q > 1$ we can try, for example, to

- reduce infeasibility by a phase-1 type pivot selection limited to rows in \mathcal{H}_2 , or
 - improve the objective function, or
 - select i with minimum row count to minimize the reduction of available columns (this choice does not require the computations of (9.123)).
- 3 The nonzero pattern of β can symbolically be updated as shown in Figure 3.3. If it is not sufficient because (9.123) is used then: Let p denote the selected pivot row. Create an elementary transformation matrix \mathbf{E} from column q using pivot element a_q^p and update β by $\beta := \mathbf{E}\beta$. In either case, update the active submatrix as in Step 6(b) of CRASH(LTSF) to ensure triangularity. Redefine \mathcal{H}_1 by (9.122) and return to step 1.

There are a few remarks to be made about CRASH(ADG).

- 1 The operation of CRASH(ADG) is not strictly symbolic. Though it uses only structural information from \mathbf{A} , the numerical updating of β (and the possible use of (9.123)), if needed, involves computations.
- 2 The complexity of the algorithm is very low therefore its operation is very fast. Indeed, if there are too many degenerate rows it probably

can do little and terminates very quickly. In the extreme case when $\mathbf{b} = \mathbf{0}$, it simply exits since \mathcal{H}_1 is empty at the outset.

- 3 If it did not do too much then a CRASH(LTSF) can follow it starting from the basis created by CRASH(ADG). Both procedures are very fast and the latter can use the data structure created for the former.

CRASH(ADG) is not a general purpose starting procedure. It can be useful if the reduction of degeneracy is important and $\mathbf{b} \neq \mathbf{0}$. It is best used in combination with other starting procedures as hinted above.

9.8.3 CPLEX basis

The complex requirements of a starting basis have motivated several other approaches. One of them has been proposed by Bixby in [Bixby, 1992] which is referred to as CPLEX basis. The main goal is formulated as to create a sparse and well behaved basis with as few type-0 variables in it as possible. Further targets are good objective value and triangularity. In this section we summarize the underlying procedure.

The problem is assumed to be in CF-1 and scaled so that it is finished by equilibration. In this way the maximum absolute value in every nonempty row and column is equal to 1. The logical variables need to be distinguished from the structurals. Therefore, the number of structural variables will, temporarily, be denoted by n .

Type-2 logical variables are included in the basis in their natural position (i.e., x_{n+i} will be the i -th basic variable if $\text{type}(x_{n+i}) = 2$). The idea behind it is that such variables naturally contribute to the sparsity of the basis and also have good numerical properties. The structural variables are assigned a preference order of inclusion in the basis. First, four sets are defined

$$\begin{aligned}\mathcal{C}_1 &= \{j : j > n, \text{type}(x_j) = 2\}, \\ \mathcal{C}_2 &= \{j : \text{type}(x_j) = 3\}, \\ \mathcal{C}_3 &= \{j : j \leq n, \text{type}(x_j) = 2\} \\ \mathcal{C}_4 &= \{j : \text{type}(x_j) = 1\}\end{aligned}$$

It is assumed there are no type-0 structurals. Therefore, every variable falls into one of the above sets. Any variable in set \mathcal{C}_i is preferred to all variables in \mathcal{C}_{i+1} , $i = 1, 2, 3$. It means that type-2 logicals are preferred to free structurals even if the latter will probably enter the basis during the simplex iterations.

For all but the type-0 variables a penalty \hat{g}_j is defined by

$$\hat{g}_j = \begin{cases} 0, & \text{if } j \in \mathcal{C}_2 \cup \mathcal{C}_3, \\ -u_j, & \text{if } j \in \mathcal{C}_4. \end{cases}$$

Let c denote the largest absolute objective coefficient of the structural variables, $c = \max\{|c_j| : 1 \leq j \leq n\}$ and define

$$c_{\max} = \begin{cases} 1000c, & \text{if } c \neq 0, \\ 1, & \text{otherwise.} \end{cases}$$

Finally, the penalty on the structural variables is defined as

$$g_j = \hat{g}_j + c_j/c_{\max}.$$

The penalties are used to sort the variables within \mathcal{C}_2 , \mathcal{C}_3 and \mathcal{C}_4 in ascending order of g_j . The lists are then concatenated into a single ordered set $\mathcal{C} = \{j_1, \dots, j_n\}$. This ranking is similar to the one used in CRASH(LTSF) in the sense that variables with the largest feasibility range are in the front of the preference list. The difference is that now the objective coefficient is used to resolve ties.

After the above preliminaries the basis \mathcal{B} is constructed by the following steps. The description uses some m -dimensional auxiliary vectors, \mathbf{h} , \mathbf{r} and \mathbf{v} . They are defined ‘on-the-fly’ as they are used.

Step 1. Include all type-2 logical variables in the basis in their home position, i.e., set $h_i = 1$ and $r_i = 1$ if $\text{type}(x_{n+i}) = 2$. Remember, the original index of the variable in basic position i is denoted by k_i .

Set $\mathcal{B} = \{k_i = n + i : \text{type}(x_{n+i}) = 2\}$. These are referred to as assigned basic variables.

For the unassigned basic positions set $h_i = 0$ and $r_i = 0$.

Set $v_i = +\infty$, $i = 1, \dots, m$.

Step 2. Let $\mathcal{C} = \{j_1, \dots, j_n\}$. For $k = 1, \dots, n$ apply the following procedure:

- (a) Let $\psi = \max\{|a_{j_k}^i|\}$. If $\psi \geq 0.99$ (reminder: the problem is assumed scaled with equilibration at the end) then:

Let i' be such that $\psi = |a_{j_k}^{i'}|$ and $r_{i'} = 0$.

Set

$$\mathcal{B} := \mathcal{B} \cup \{j_k\},$$

$$h_{i'} = 1,$$

$v_{i'} = \psi$, and

$r_i := r_i + 1$ for all i such that $a_{j_k}^i \neq 0$.

Increment k and return to the beginning of (a).

- (b) If $|a_{j_k}^i| > 0.01v_i$ for some i then go to the beginning of (a) with the next value of k . Otherwise:

Let $\psi = \max\{|a_{j_k}^i| : h_i = 0\}$. If $\psi = 0$, increment k and go to the beginning of (a). Otherwise:

Let i' such that $h_{i'} = 0$ and $\psi = |a_{j_k}^{i'}|$.

Set

$\mathcal{B} := \mathcal{B} \cup \{j_k\}$,

$h_{i'} = 1$,

$v_{i'} = \psi$, and

$r_i := r_i + 1$ for all i such that $a_{j_k}^i \neq 0$.

Increment k and return to the beginning of (a).

Step 3. If there are unassigned basic positions, include the corresponding logical variable in the basis. Formally, for all remaining i such that $h_i = 0$, set $\mathcal{B} := \mathcal{B} \cup \{n + i\}$.

The above procedure has some magic numbers that can help do certain things. For instance, condition $|a_{j_k}^i| \leq 0.01v_i$ in Step 2(b) helps enforce an approximate lower triangularity of the basis matrix. This is because if 0.01 is replaced by 0 and \mathbf{v} is initialized to any finite value for the type-2 logicals then the resulting \mathbf{B} is lower triangular. Similarly, if \mathbf{v} is initialized to 1 for the type-2 logicals and 0.99 is replaced by 1 in Step 2(a) then all v_i values are equal to 1.

The implementation of the CPLEX basis procedure is quite simple and its operation is fast. It is considered a good average performer worth including in an advanced system to increase the choice of starting procedures.

In the original description of the CPLEX basis the explicit lower bounds of the variables are preserved and used to refine the penalties. If an implementation uses the CF-2 without bound translation then the description detailed in [Bixby, 1992] (which is somewhat more complex than the one given here) has to be followed.

9.8.4 A tearing algorithm

Gould and Reid have proposed a very remarkable starting procedure for large and sparse systems in [Gould and Reid, 1989]. Their main purpose is to find an initial basis that is as feasible as possible and, of course, can be obtained by a reasonable computational effort. The authors call the method they have developed *tearing algorithm*.

The algorithm is based on two main assumptions. The first is that matrix \mathbf{A} can be reordered into a lower block triangular form with small blocks and the second is that an efficient and reliable dense simplex solver (referred to as DLP) is available for solving problems with up to s rows, where s is a small number, like 5.

To be more specific, it is assumed that after suitable row and column permutations \mathbf{A} is in the following form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1^1 & & & & \\ \mathbf{A}_1^2 & \mathbf{A}_2^2 & & & \\ \vdots & \vdots & \ddots & & \\ \mathbf{A}_1^r & \mathbf{A}_2^r & \dots & \mathbf{A}_r^r & \\ \mathbf{A}_1^{r+1} & \mathbf{A}_2^{r+1} & \dots & \mathbf{A}_r^{r+1} & \mathbf{0} \end{bmatrix}. \quad (9.124)$$

Here each submatrix \mathbf{A}_j^i is $m_j \times n_j$ and the dimensions are positive numbers for $j = 1, \dots, r$, while m_{r+1} and n_{r+1} may be zero. The issue of determining the permutations needed to obtain the partitioning of (9.124) is deferred for later.

In the rest of this section it is assumed that the LP problem is in CF-1. Currently, it is also assumed that \mathbf{A} has been rearranged into the form of (9.124). We can augment \mathbf{A} so that a feasible solution is easily obtained to the augmented problem. This is a kind of a goal programming approach where a nonnegative ‘undershoot’ and ‘overshoot’ variable is assigned to each constraint such that

$$\mathbf{a}^i \mathbf{x} + v_i - w_i = b_i, \quad v_i, w_i \geq 0,$$

or in matrix form

$$\mathbf{Ax} + \mathbf{v} - \mathbf{w} = \mathbf{b}, \quad \mathbf{v}, \mathbf{w} \geq \mathbf{0}. \quad (9.125)$$

In a feasible solution to $\mathbf{Ax} = \mathbf{b}$ (plus the type constraints) $\mathbf{v} = \mathbf{w} = \mathbf{0}$ and the type constraints on the \mathbf{x} variables are satisfied. To achieve this, we minimize the $\zeta = \mathbf{1}^T(\mathbf{v} + \mathbf{w})$ objective function, where $\mathbf{1}$ is a vector of 1's with appropriate dimension, and hope to reach a basic feasible solution with $\zeta = 0$. It may or may not be attained but, in any case, we obtain a basis that is closer to feasibility than the all logical basis.

(9.125) can be written out in the partitioned form as

$$\begin{bmatrix} \mathbf{A}_1^1 & & & & \\ \mathbf{A}_1^2 & \mathbf{A}_2^2 & & & \\ \vdots & \vdots & \ddots & & \\ \mathbf{A}_1^r & \mathbf{A}_2^r & \dots & \mathbf{A}_r^r & \\ \mathbf{A}_1^{r+1} & \mathbf{A}_2^{r+1} & \dots & \mathbf{A}_r^{r+1} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_r \\ \mathbf{x}_{r+1} \end{bmatrix} + \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_r \\ \mathbf{v}_{r+1} \end{bmatrix} - \begin{bmatrix} \mathbf{w}_1 \\ \mathbf{w}_2 \\ \vdots \\ \mathbf{w}_r \\ \mathbf{w}_{r+1} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_r \\ \mathbf{b}_{r+1} \end{bmatrix} \quad (9.126)$$

with $\mathbf{v}, \mathbf{w} \geq \mathbf{0}$.

From now on we assume that condition

$$m_i \leq s, \quad 1 \leq i \leq r \quad (9.127)$$

holds. The first block of (9.126) and the type constraint on variables in \mathbf{x}_1 require that

$$\mathbf{A}_1^T \mathbf{x}_1 + \mathbf{v}_1 - \mathbf{w}_1 = \mathbf{b}_1, \quad \mathbf{v}_1, \mathbf{w}_1 \geq \mathbf{0}, \quad \text{and type constraints on } \mathbf{x}_1 \quad (9.128)$$

are satisfied. Now we want to drive both \mathbf{v}_1 and \mathbf{w}_1 to zero. Due to condition (9.127) we may achieve this by using DLP to minimize $\mathbf{1}^T(\mathbf{v}_1 + \mathbf{w}_1)$ subject to (9.128). The solution of this problem produces the optimal objective value and solution vectors $\hat{\mathbf{x}}_1, \hat{\mathbf{v}}_1, \hat{\mathbf{w}}_1$, together with a set of m_1 basic variables which may include some variables from \mathbf{v} and \mathbf{w} . If the original problem has a feasible solution then for the first block we obtain $\hat{\mathbf{v}}_1 = \hat{\mathbf{w}}_1 = \mathbf{0}$.

Moving down the diagonal blocks of (9.124), at stage k we have the solution vectors $\hat{\mathbf{x}}_i, \hat{\mathbf{v}}_i, \hat{\mathbf{w}}_i$ for $1 \leq i < k$ and a set of $m_1 + \dots + m_{k-1}$ basic variables. At stage k we want to satisfy

$$\mathbf{A}_k^T \mathbf{x}_k + \mathbf{v}_k - \mathbf{w}_k = \mathbf{b}_k - \sum_{i=1}^{k-1} \mathbf{A}_i^T \hat{\mathbf{x}}_i, \quad \mathbf{v}_k, \mathbf{w}_k \geq \mathbf{0}, \quad \text{and type constraints on } \mathbf{x}_k \quad (9.129)$$

so that $\hat{\mathbf{v}}_k = \hat{\mathbf{w}}_k = \mathbf{0}$. Such a solution is called a *satisfactory solution* to (9.129). A satisfactory solution may be obtained by minimizing $\zeta_k = \mathbf{1}^T(\mathbf{v}_k + \mathbf{w}_k)$ subject to the constraints in (9.129). As $m_k \leq s$, DLP can be used to solve this problem. The optimal solution gives m_k basic variables together with the solution vectors $\hat{\mathbf{x}}_k, \hat{\mathbf{v}}_k, \hat{\mathbf{w}}_k$ which are then passed over to stage $k+1$.

This procedure is applied for stages $k = 2, 3, \dots, r$. As the $(r+1)$ st diagonal block is $\mathbf{0}$, \mathbf{x}_{r+1} cannot contribute to satisfy the corresponding LP problem. Therefore, the \mathbf{v}_{r+1} and \mathbf{w}_{r+1} variables have to be used to match \mathbf{b}_{r+1} . This is achieved by setting

$$\hat{\mathbf{v}}_{r+1} = \max \left\{ \mathbf{0}, \mathbf{b}_{r+1} - \sum_{i=1}^r \mathbf{A}_i^{r+1} \hat{\mathbf{x}}_i \right\} \quad (9.130)$$

and

$$\hat{\mathbf{w}}_{r+1} = \max \left\{ \mathbf{0}, -\mathbf{b}_{r+1} + \sum_{i=1}^r \mathbf{A}_i^{r+1} \hat{\mathbf{x}}_i \right\} \quad (9.131)$$

where the maximum is taken by coordinates. Variables in $\hat{\mathbf{v}}_{r+1}$ and $\hat{\mathbf{w}}_{r+1}$ with nonzero value are chosen to be basic and further ones can be

picked arbitrarily for the remaining (uncovered) rows to make up a full basis.

It may happen that a satisfactory solution cannot be found for the subproblem at stage k . It could be the consequence of inappropriate setting of variables \hat{x}_i , $i = 1, \dots, k - 1$. In such a case the following backtracking can be done. If the sizes of the preceding blocks satisfy $m_j + m_{j+1} + \dots + m_k \leq s$ with some j then subproblems $j, j+1, \dots, k$ can be viewed as one and the corresponding problem solved by DLP. This increased flexibility may result in a satisfactory solution at the expense of discarding the previously calculated solutions of the preceding subproblems and solving a larger LP. Though the size of this LP is still within the capabilities of DLP the computational cost of a backtrack is definitely substantial.

It remains to see how a block lower-triangular form of (9.124) can be obtained. From among the several possibilities we briefly outline a promising one which is an adapted version of the P5 algorithm of Erisman, Grimes, Lewis and Poole [Erisman et al., 1985]. It uses column and row permutations.

P5 requires the setting up and updating of counts of nonzeros by rows and columns. It also uses the notion of the active submatrix which is the entire matrix at the beginning. First the minimum row count, denoted by ν_{\min} is identified. It may not be uniquely determined. Next, every column is considered which has a nonzero in at least one row for which the row count is ν_{\min} . P5 chooses a column from among these which has the maximum number of nonzeros in positions belonging to rows with the minimum row count. If several columns satisfy this condition then one with maximum column count is chosen (unless the number of entries with minimum row count is one). This column is permuted to the next available unpermuted position (at the beginning it is the first). The rows with the current minimum row count are permuted up to the next available positions. If $\nu_{\min} = 1$ then the participating columns are taken in any order. If a column is moved to its final position the row counts are updated. With appropriate data structures the operation of this procedure will be very fast. It is easy to see that in the resulting permuted matrix the diagonal blocks will be fully dense.

If there is a block for which $m_k > s$ then DFL cannot be used to solve subproblem k . In this case (9.129) needs to be partitioned rowwise further into two

$$\mathbf{A}_k^{k1} \mathbf{x}_k + \mathbf{v}_{k1} - \mathbf{w}_{k1} = \mathbf{b}_{k1} - \sum_{i=1}^{k-1} \mathbf{A}_k^{i1} \hat{\mathbf{x}}_i, \quad \mathbf{v}_{k1}, \mathbf{w}_{k1} \geq \mathbf{0}, \quad (9.132)$$

and

$$\mathbf{A}_k^{k2} \mathbf{x}_k + \mathbf{v}_{k2} - \mathbf{w}_{k2} = \mathbf{b}_{k2} - \sum_{i=1}^{k-1} \mathbf{A}_k^{i2} \hat{\mathbf{x}}_i, \quad \mathbf{v}_{k2}, \mathbf{w}_{k2} \geq \mathbf{0}, \quad (9.133)$$

where \mathbf{A}_k^{k1} has s rows and as large a rank as possible. This can be achieved by reordering the rows of \mathbf{A}_k^k . For problem in (9.132) we can apply DLP in an attempt to obtain a satisfactory solution. The resulting vector $\hat{\mathbf{x}}_k$ can be used to determine $\hat{\mathbf{v}}_{k2}, \hat{\mathbf{w}}_{k2}$ in a similar way as $\hat{\mathbf{v}}_{r+1}, \hat{\mathbf{w}}_{r+1}$ in (9.130) and (9.131)

$$\hat{\mathbf{v}}_{k2} = \max \left\{ \mathbf{0}, \mathbf{b}_{k2} - \sum_{i=1}^r \mathbf{A}_k^{i2} \hat{\mathbf{x}}_i \right\}$$

and

$$\hat{\mathbf{w}}_{k2} = \max \left\{ \mathbf{0}, -\mathbf{b}_{k2} + \sum_{i=1}^r \mathbf{A}_k^{i2} \hat{\mathbf{x}}_i \right\}.$$

Similarly to the $r+1$ case above, a set of $m_k - s$ basic variables can be determined for (9.133). They and the s basic variables for (9.133) are passed to stage k together with the solution vectors $\mathbf{x}_k, \hat{\mathbf{v}}_{k1}, \hat{\mathbf{w}}_{k1}, \hat{\mathbf{v}}_{k2}$ and $\hat{\mathbf{w}}_{k2}$.

There are several possibilities to refine the above tearing algorithm. Just to mention one of them, it is possible to use the already existing logical variables in place of some components of \mathbf{v} or \mathbf{w} if they can be included in the procedure at feasible level at any stage.

The tearing algorithm showed a remarkable performance on medium size sparse problems both in speed and quality of the basis when $s = 5$ was used. As s is increased the extra work may grow too fast. The same may be true if the problem size gets bigger. The performance of the method has not been tested systematically for problems with tens of thousands of rows.

The original paper [Gould and Reid, 1989] discusses some variations of the above described version of the tearing algorithm.

9.8.5 Partial basis

It is possible to try to create a basis by nominating some structural variables into the initial basis. This can be the case when the model builder has some knowledge about the problem and can give a list of ‘good’ candidate columns for inclusion. This list can be shorter, equal to or even longer than m . It can happen that different versions of a problem need to be solved. If, after the solution the size of the problem

changes the vectors of the optimal basis of the original problem are typically good candidates for inclusion in a starting basis though they probably will not make a full rank.

As the creation of an initial basis always uses \mathbf{I} as a starting point, in principle, one can attempt to replace the unit columns of \mathbf{I} by the given list of columns of structural variables.

The simplest way of accommodating this idea is to submit the list to the factorization routine and let it 'do the job'. Obviously, in this case the issues of linear independence and numerical stability are addressed automatically. The routine will use as many candidates as it can and the remaining positions will be filled by the appropriate unit vectors if necessary. (Something similar takes place when the routine is used to invert a basis that turns out to be singular.) Of course, if the list is longer than m there is no guarantee that all candidates will be considered. A further problem is that if the candidates do not form a full basis then the basic positions of the vectors can be crucial from the point of view of feasibility or optimality of the basis. However, using the factorization routine, we cannot influence this choice. This problem considerably undermines the usefulness of this approach. Attempts can be made on an individual basis but the method of partial basis is not considered a serious contender for creating an initial basis for the simplex method.

Chapter 10

THE DUAL ALGORITHM

10.1. Dual feasibility, infeasibility

In section 2.4.1, for the standard form of the LP, it was shown that the dual feasibility and primal optimality conditions are identical and the primal reduced cost vector \mathbf{d} is the same as the vector of dual logical variables. In case of CF-1 or CF-2 similar statements are true. This fact can be utilized to introduce some efficient dual algorithms.

Stating the dual of CF-1 or CF-2 would cause some notational complications as they are not in a strictly analytical form because of the type restrictions. However, we can investigate the types of dual constraints generated by the different types of variables in CF-1 and CF-2. Most of the conclusions below are direct consequences of the $(P1) - (D1)$ or $(P2) - (D2)$ primal-dual pairs introduced in section 2.3. To point out the high level of symmetry between primal and dual even in the general form, we use CF-2 to include minus type variables. It should be remembered that the primal is a minimization and, therefore, the dual is a maximization problem. The correspondence between the primal variables and dual constraints is summarized in the following table.

Primal variable	Dual constraint	
$x_j = 0$	$\mathbf{a}_j^T \mathbf{y} <> c_j$	
$x_j \geq 0$	$\mathbf{a}_j^T \mathbf{y} \leq c_j$	(10.1)
$x_j \leq 0$	$\mathbf{a}_j^T \mathbf{y} \geq c_j$	
x_j free	$\mathbf{a}_j^T \mathbf{y} = c_j$	

Note, symbol $<>$ refers to a non-binding constraint and \mathbf{a}_j is the j -th column of \mathbf{A} . With the help of an appropriate dual logical variable each

of these dual constraints can be converted into an equality.

Primal variable	Dual constraint	Dual logical
$x_j = 0$ type-0	$\mathbf{a}_j^T \mathbf{y} + d_j = c_j$	d_j free, type-3
$x_j \geq 0$ type-2	$\mathbf{a}_j^T \mathbf{y} + d_j = c_j$	$d_j \geq 0$, type-2
$x_j \leq 0$ type-4	$\mathbf{a}_j^T \mathbf{y} + d_j = c_j$	$d_j \leq 0$, type-4
x_j free type-3	$\mathbf{a}_j^T \mathbf{y} + d_j = c_j$	$d_j = 0$, type-0

The case of the remaining type-1 primal variables is a bit different. An individual bound $0 \leq x_j \leq u_j$ can be included in the general constraints as $x_j + \xi_j = u_j$ with $x_j, \xi_j \geq 0$. It means that x_j can be treated as a type-2 variable. As there is one more constraint now, one more variable is defined in the dual and the j -th dual constraint becomes $\mathbf{a}_j^T \mathbf{y} - w_j \leq c_j$, with $w_j \geq 0$. It can be converted into an equality by the addition of a type-2 dual logical variable d_j : $\mathbf{a}_j^T \mathbf{y} - w_j + d_j = c_j$, $d_j \geq 0$. However, if a nonbasic x_j is at its upper bound its feasible displacement is negative (nonpositive), i.e., it can be treated as a type-4 variable. In this case, according to (10.1), a ‘ \geq ’ type dual constraint is generated that can be converted into equality by the addition of a nonpositive dual logical variable d_j as $\mathbf{a}_j^T \mathbf{y} - w_j + d_j = c_j$, $d_j \leq 0$.

The above derivation can be viewed as a detailed outline of the primal-dual relationship in the case of type-1 primal variables. A more rigorous investigation results, obviously, in the same conclusion.

10.2. Improving a dual feasible solution

The primal optimality conditions for the computational forms were given in (9.16) and (9.37). Now we can use them directly as conditions of dual feasibility. In the rest of this chapter CF-1 will be considered except when we point out the changes needed to cover CF-2, as well.

For easier reference, the primal optimality conditions for CF-1 are restated here. In the following table, x_j denotes a nonbasic primal variable and d_j is its reduced cost, or equivalently, the dual logical variable corresponding to the j -th nonbasic dual constraint.

Type(x_j)	Value	d_j	Remark
0	$x_j = 0$	Immaterial	
1	$x_j = 0$	≥ 0	
1	$x_j = u_j$	≤ 0	$j \in \mathcal{U}$
2	$x_j = 0$	≥ 0	
3	$x_j = 0$	$= 0$	

(10.2)

It is immediately obvious that dual logical variables corresponding to type-0 primal variables are always at feasible level, therefore, they can be left out from further investigations of the dual.

Assume an index set \mathcal{U} of nonbasic variables at upper bound and a basis \mathcal{B} are given so that dual feasibility is satisfied. Recall, from (2.52), (2.58) and (2.59), the dual solution is $\mathbf{y}^T = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1}$, $\mathbf{d}_{\mathcal{B}}^T = \mathbf{0}^T$ and $\mathbf{d}_{\mathcal{R}}^T = \mathbf{c}_{\mathcal{R}}^T - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{R}$. If the corresponding primal solution from (9.7) (or (9.8)), $\beta = \mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1} \mathbf{b}_{\mathcal{U}}$, is feasible then it is optimal. Otherwise, the current solution is primal infeasible. Thus, there exists a neighboring dual feasible basis with better (or not worse) dual objective value unless dual unboundedness is detected.

Section 2.4.2 described how a non-optimal dual basic feasible solution can be improved in case of the standard form of LP (with only type-2 variables). The main idea was the relaxation of the p -th dual basic equation such that the dual objective could improve. For this purpose, a basic equation corresponding to an infeasible primal basic variable was chosen. Arguments used there can easily be generalized for the case when there are other types of variables in the problem. Equations (2.67) and (2.68) showed that the change of the dual solution and objective as the function of t (the level of relaxation) can be expressed as $\mathbf{y}(t) = \mathbf{y} + t\boldsymbol{\rho}_p$ and $Z(t) = Z(0) + t\beta_p$, respectively, where $\boldsymbol{\rho}_p$ denotes the p -th row of \mathbf{B}^{-1} and $\beta_p = x_{k_p}$ is the p -th component of the primal basic solution.

Recalling further notations used in Section 2.4.2, the p -th dual basic equation corresponds to variable x_{k_p} and $\mu = k_p$. If, formally, we write $\mathbf{a}_{\mu}^T \mathbf{y}(t) - t = c_{\mu}$ then $-t$ is the value of the μ -th dual logical variable which must be dual feasible. The dual logicals change their value according to (2.70), i.e., $d_j(t) = c_j - \mathbf{a}_j^T \mathbf{y}(t)$, $j \in \mathcal{R}$ and $j \neq \mu$. As Z is to be maximized, we need

$$\Delta Z = Z(t) - Z(0) = t\beta_p \geq 0. \quad (10.3)$$

In the case of the standard LP $t \leq 0$ was necessary. Therefore, $\Delta Z \geq 0$ could be achieved only with $\beta_p < 0$, i.e., if a primal infeasible basic position was chosen. There are more possibilities if all types of variables are present.

10.2.1 Dual algorithm with type-1 and type-2 variables

First, we investigate the situation when there are only type-1 and type-2 structural variables in the LP problem. Note, in this case a basic solution is determined by \mathcal{B} and \mathcal{U} .

Dual feasibility of type-1 variables means that $d_j \geq 0$ if x_j is nonbasic at zero and $d_j \leq 0$ if x_j is nonbasic at upper bound. Additionally,

an upper bounded basic variable can be infeasible in two ways, either $\beta_i < 0$ or $\beta_i > v_i$, where v_i , as before, denotes the upper bound of β_i . Therefore, if a $\beta_p < 0$ is chosen to leave the basis then $t \leq 0$ is required by (10.3) and if $\beta_p > v_p$ is selected then $t \geq 0$ must hold. It also entails that the outgoing variable must leave the basis at lower bound if $t \leq 0$ or upper bound if $t \geq 0$.

The displacement of the dual logical variable of the relaxed equation must be determined in such a way that (i) this logical variable takes a dual feasible value ($-t$ must be dual feasible) and (ii) all other dual logicals remain dual feasible after the transformation, i.e.,

$$d_j(t) = d_j - t\alpha_j^p, \quad j \in \mathcal{R},$$

must be feasible.

If $t \leq 0$ is required then:

- As $d_j(t) = d_j - t\alpha_j^p \geq 0$ must hold for $j \in (\mathcal{R} \setminus \mathcal{U})$ (now $d_j \geq 0$), a lower bound on t is obtained if $\alpha_j^p < 0$, which is $t \geq d_j/\alpha_j^p$. On the other hand, $d_j(t)$ remains feasible if $\alpha_j^p > 0$.
- As $d_j(t) = d_j - t\alpha_j^p \leq 0$ must hold for $j \in \mathcal{U}$ (note, $d_j \leq 0$), a lower bound on t is obtained if $\alpha_j^p > 0$, namely $t \geq d_j/\alpha_j^p$. Obviously, $d_j(t)$ remains feasible if $\alpha_j^p < 0$.

Define two index sets, $\mathcal{J}_1 = \{j : \alpha_j^p < 0, j \in (\mathcal{R} \setminus \mathcal{U})\}$ and $\mathcal{J}_2 = \{j : \alpha_j^p > 0, j \in \mathcal{U}\}$ and let $\mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2$. These are the positions that determine the maximum relaxation t of the p -th basic dual equation such that some other dual constraint becomes tight (its dual logical becomes zero). It is determined by the dual ratio test

$$\theta_D = \frac{d_q}{\alpha_q^p} = \max_{j \in \mathcal{J}} \left\{ \frac{d_j}{\alpha_j^p} \right\} = \min_{j \in \mathcal{J}} \left\{ \left| \frac{d_j}{\alpha_j^p} \right| \right\}. \quad (10.4)$$

If $t \geq 0$ is required then:

- As $d_j(t) = d_j - t\alpha_j^p \geq 0$ must hold for $j \in (\mathcal{R} \setminus \mathcal{U})$ (now $d_j \geq 0$), an upper bound on t is obtained if $\alpha_j^p > 0$, which is $t \leq d_j/\alpha_j^p$. On the other hand, $d_j(t)$ remains feasible if $\alpha_j^p < 0$.
- As $d_j(t) = d_j - t\alpha_j^p \leq 0$ must hold for $j \in \mathcal{U}$ (note, $d_j \leq 0$), an upper bound on t is obtained if $\alpha_j^p < 0$, namely $t \leq d_j/\alpha_j^p$. Obviously, $d_j(t)$ remains feasible if $\alpha_j^p > 0$.

Define two index sets, $\mathcal{J}_1 = \{j : \alpha_j^p > 0, j \in (\mathcal{R} \setminus \mathcal{U})\}$ and $\mathcal{J}_2 = \{j : \alpha_j^p < 0, j \in \mathcal{U}\}$ and let $\mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2$. Again, these are the positions that

determine the maximum relaxation of the p -th basic dual equation such that the dual logical variable of some nonbasic dual constraint becomes 0. It is determined by the dual steplength

$$\theta_D = \frac{d_q}{\alpha_q^p} = \min_{j \in \mathcal{J}} \left\{ \frac{d_j}{\alpha_j^p} \right\} \quad \text{which is formally equal to} \quad \min_{j \in \mathcal{J}} \left\{ \left| \frac{d_j}{\alpha_j^p} \right| \right\}.$$

Based on the above observations we can define a dual algorithm for an LP problem with type-1 and type-2 variables. We call it *Dual-U* algorithm. Following Chvatal [Chvátal, 1983], the steps of Dual-U are given below.

Algorithm Dual-U

Assume, the LP problem is in CF-1 but only type-1 and type-2 structurals are present. \mathcal{B} and \mathcal{U} are given and define a dual feasible solution.

Step 1. If the primal solution is feasible, that is, $\mathbf{0} \leq \boldsymbol{\beta} \leq \boldsymbol{v}$ then it is optimal. Otherwise, there is at least one primal infeasible basic variable. Any such variable can be selected to leave the basis. Let the basic position of the chosen one be denoted by p so that the leaving variable is $\beta_p = x_\mu$ with $\mu = k_p$.

Step 2. Determine the nonbasic components of the updated pivot row: $\alpha_{\mathcal{R}}^p = \rho_p^T \mathbf{R}$.

Step 3. Determine the index set of eligible pivot positions. If $\beta_p < 0$ then let $\mathcal{J}_1 = \{j : \alpha_j^p < 0, j \in (\mathcal{R} \setminus \mathcal{U})\}$ and $\mathcal{J}_2 = \{j : \alpha_j^p > 0, j \in \mathcal{U}\}$. If $\beta_p > v_p$ then let $\mathcal{J}_1 = \{j : \alpha_j^p > 0, j \in (\mathcal{R} \setminus \mathcal{U})\}$ and $\mathcal{J}_2 = \{j : \alpha_j^p < 0, j \in \mathcal{U}\}$. In both cases, let $\mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2$.

If \mathcal{J} is empty then stop: the problem is dual unbounded and, hence, primal infeasible. Otherwise, find $q \in \mathcal{J}$ such that

$$\theta_D = \left| \frac{d_q}{\alpha_q^p} \right| = \min_{j \in \mathcal{J}} \left| \frac{d_j}{\alpha_j^p} \right|. \quad (10.5)$$

It defines the entering variable to be x_q .

Step 4. Determine the transformed entering column: $\alpha_q = \mathbf{B}^{-1} \mathbf{a}_q$.

Step 5. Update solution.

Set $\theta_P = \frac{\beta_p}{\alpha_q^p}$ and $\bar{x}_\mu = 0$ in case $\beta_p < 0$ or $\theta_P = \frac{\beta_p - v_p}{\alpha_q^p}$ and $\bar{x}_\mu = u_\mu$ in case $\beta_p > v_p$.

Update the basic solution: $\bar{\beta}_i = \beta_i - \theta_P \alpha_q^i$ for $1 \leq i \leq m$, $i \neq p$ and
 $\bar{\beta}_p = x_q + \theta_P$. (10.6)

Note, the incoming variable may enter the basis at infeasible level.

Update dual logicals of nonbasic variables: $\bar{d}_\mu = -\frac{d_q}{\alpha_q^p}$ and $\bar{d}_j = d_j + \bar{d}_\mu \alpha_j^p$ for $j \neq \mu$.

Go to Step 1.

10.2.2 Dual algorithm with all types of variables

Nonbasic type-0 variables do not play any role in the dual method since any d_j is feasible for a fixed variable x_j . Such variables do not take part in the dual ratio test (are not included in \mathcal{J}) and, therefore, they never enter the basis. Note, this is completely in line with the selection principles of the primal simplex algorithm. If a type-0 variable is basic (e.g., the logical variable of an equality constraint) it can be feasible (i.e., equal to zero) or infeasible (positive or negative). Such an infeasible variable is always a candidate to leave the basis and thus improve the dual objective as it can be seen from (10.3). Since the reduced cost of the outgoing type-0 variable is unrestricted in sign it can always be chosen to point to the improving direction of the dual objective function. This and the next subsection are based on papers by Fourer [Fourer, 1994] and Maros [Maros, 2002b].

The way type-1 variables can be handled has already been discussed. In many, but not all, aspects type-0 variables can be viewed as special cases of type-1 when $\ell_j = u_j = 0$. The main difference is that once a type-0 variable left the basis it will never be a candidate to enter again.

It is easy to see that Dual-U algorithm requires very few changes to take care of all types of variables. Type-0 variables are of interest only if they are basic at infeasible level. At the other extreme, type-3 variables are involved in the dual algorithm as long as they are nonbasic. In this case they need to be included in the ratio test as their d_j must remain zero. They will certainly take part in the (10.5) dual ratio test if the corresponding $\alpha_j^p \neq 0$. Since in this case the ratio is zero such a variable will be selected to enter the basis. This, again, is very similar to the tendency of the primal method which also favors free variables to become basic if their reduced cost is nonzero.

For the formal statement of the algorithm with all types of variables, which we refer to as *Dual-G* (Dual General), only Step 3 of Dual-U has to be adjusted. It is important to remember that now the basis may contain some type-0 variables for which both the lower and upper bounds

are zero. Therefore, such basic variables can be infeasible positively or negatively the same way as type-1 variables. If selected, a type-0 variable will leave the basis at zero level but the sign of its d_j will become unimportant after the iteration as it is unrestricted.

Algorithm Dual-G

All steps are identical with the steps of Algorithm Dual-U except Step 3 which becomes the following.

Step 3. Let \mathcal{J}_1 be the set of indices of type-3 nonbasic variables for which $\alpha_j^p \neq 0$.

For type-1 and type-2 nonbasic variables:

If $\beta_p < 0$ then let \mathcal{J}_2 be the index set of those x_j s for which $\alpha_j^p < 0$ and $x_j = 0$, or $\alpha_j^p > 0$ and $x_j = u_j$. If $\beta_p > v_p$ then let \mathcal{J}_2 be the set of those x_j s for which $\alpha_j^p > 0$ and $x_j = 0$, or $\alpha_j^p < 0$ and $x_j = u_j$.

Let $\mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2$.

If \mathcal{J} is empty then stop. The problem is dual unbounded and, hence, primal infeasible.

Otherwise, find $q \in \mathcal{J}$ such that

$$\theta_D = \left| \frac{d_q}{\alpha_q^p} \right| = \min_{j \in \mathcal{J}} \left| \frac{d_j}{\alpha_j^p} \right|$$

and let x_q be the entering variable (θ_D is the selected dual ratio).

10.2.3 Bound flip in dual

While Dual-U and Dual-G are perfectly legitimate algorithms for the case when upper bounded variables are present in a problem further investigations reveal some additional possibilities that can be computationally appealing.

At the beginning of an iteration the incoming variable x_q is nonbasic at a feasible level. It becomes basic in position p with a value of $x_q + \theta_P$. This new value can be infeasible if x_q is a bounded variable and its displacement is greater than its own individual bound, i.e.,

$$|\theta_P| > u_q. \quad (10.7)$$

Though this situation does not change the theoretical convergence properties of the dual algorithm (no basis is repeated if dual degeneracy is treated properly), computationally it is usually not advantageous.

The introduction of a new infeasible basic variable can be avoided. One possibility is to set x_q to its opposite bound, i.e., perform a bound

flip. It entails the updating of the basic solution. However, in this case the corresponding dual logical variable becomes infeasible as d_q remains unchanged (see (10.2)). At the same time, the updated β_p , as will be shown, remains infeasible and can be selected again. The dual ratio test can be performed at no cost as the updated pivot row remains available. This step can be repeated recursively until a variable is found that enters the basis at a feasible level or we can conclude that the dual solution is unbounded. In the former case the dual feasibility of variables participating in bound change is automatically restored. A careful analysis shows that this idea leads to an algorithm that has some remarkable features. One of them is that its local efficiency just increases with the number of upper bounded variables.

Bounded (type-0 or type-1) basic variables can be infeasible in two different ways. These cases will be discussed separately followed by a framework that combines them in a unified algorithm.

First, we assume that p was selected because $\beta_p < 0$. In this case the p -th dual constraint is released negatively and the change of the objective function is described by (2.68) which says $Z(t) = Z(0) + t\beta_p$. Now the ratios are defined by (10.4) (which is equivalent with (10.5)) and are negative. If the ratio test selects an incoming variable x_q and (10.7) holds for the resulting $\theta_P = \beta_p/\alpha_q^p$ then a bound flip of x_q is triggered and the β basic solution is updated as $\beta = \beta \pm u_q \alpha_q$. More specifically, the selected infeasible basic variable gets updated as $\bar{\beta}_p = \beta_p \pm u_q \alpha_q^p$. Here, ‘−’ applies if $x_q = 0$, and ‘+’ if $x_q = u_q$ before the iteration. Taking into account the definition of \mathcal{J} (see Step 3 of Dual-G) the two cases can be expressed by the single equation:

$$\bar{\beta}_p = \beta_p + |\alpha_q^p| u_q.$$

Here, $\bar{\beta}_p < 0$ holds because $\beta_p < 0$ and, by assumption, $|\beta_p/\alpha_q^p| > u_q$. Therefore, the originally selected infeasible basic variable remains infeasible. It implies that it can be selected again and the dual objective function keeps improving if t moves beyond θ_D , however, the rate of improvement decreases by $|\alpha_q^p| u_q$ from $-\beta_p$ to $-\beta_p - |\alpha_q^p| u_q$. Now, we leave out index q from \mathcal{J} , go to the (10.5) ratio test, and repeat the above procedure. This can easily be done since the updated p -th row remains available unchanged. If with the newly defined θ_P we find $|\theta_P| \leq u_q$ then the corresponding x_q is the incoming variable and a basis change is performed. If $\bar{\beta}_p < 0$ holds and \mathcal{J} becomes empty then the dual is unbounded and the primal is infeasible.

Figure 10.1 shows the change of the dual objective as a function of relaxing the p -th dual constraint. This is a piecewise linear function of t . The first break point corresponds to the smallest ratio the second to

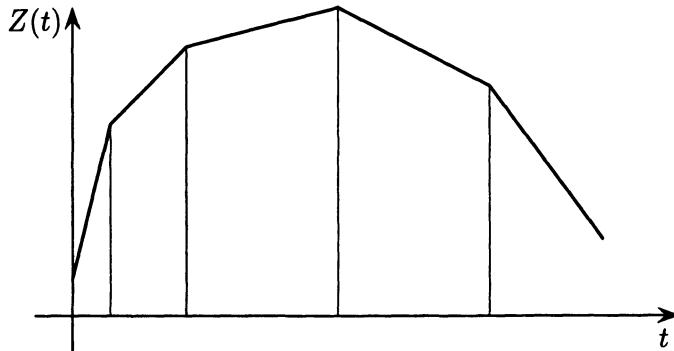


Figure 10.1. The dual objective as a function of relaxing the p -th dual basic equation by t .

the second smallest, and so on. At the maximum the sign of the slope of the function changes. It is determined by $|\theta_P| \leq u_q$, that is, when a basis change is performed.

Next, we consider the case when p was selected because $\beta_p > v_p$. Any constraint $0 \leq x \leq u$ defining an upper bounded variable can be written as $x + w = u$, $x, w \geq 0$, where w is a slack variable. Therefore, the $\beta_p > v_p$ condition is equivalent to $w = u - x < 0$. This latter indicates that if the signs of the entries in row p are changed we have the same situation as before. Therefore, by relaxing the μ -th dual constraint positively to $\mathbf{a}_\mu^T \mathbf{y} \geq c_\mu$ (which is now parameterized by $t \geq 0$) the change in the dual objective is $\Delta Z = -tw = t(\beta_p - v_p)$, that is, the rate of change is $\beta_p - v_p > 0$ as t moves away from 0 positively. The $\mathbf{a}_\mu^T \mathbf{y} \geq c_\mu$ type relaxation means that the corresponding primal variable must leave the basis at upper bound (see (10.2)).

Similarly to the first case, (10.5) is performed with the appropriate \mathcal{J} . Note, the ratios are positive now. A bound flip of x_q is triggered if (10.7) holds and the updated value of the p -th basic variable becomes $\bar{\beta}_p = \beta_p \pm u_q \alpha_{pq}$. Here, '+' applies if $x_q = 0$, and '-' if $x_q = u_q$ before the iteration, but in this case (see definition of \mathcal{J} in Dual-G) the single updating equation is

$$\bar{\beta}_p = \beta_p - |\alpha_q^p| u_q.$$

Now, $\bar{\beta}_p > v_p$ holds because of $\theta_P = |(\beta_p - v)/\alpha_q^p| > u_q$, meaning that the originally selected infeasible basic variable remains infeasible. It implies that the dual objective function keeps improving if t is increased beyond θ_D , however, the rate of improvement decreases again by $|\alpha_q^p| u_q$. The rest of the arguments of the first case applies here unchanged.

The important common feature of the two cases is that the slope of the corresponding piecewise linear function (the dual objective) decreases at every bound flip by $|\alpha_q^p|u_q$.

In other words, the initial slope of the dual objective is

$$s_0 = \begin{cases} -\beta_p, & \text{if } \beta_p < 0 \\ \beta_p - v_p, & \text{if } \beta_p > v_p. \end{cases} \quad (10.8)$$

This slope changes at the k -th bound flip to

$$s_k = s_{k-1} - |\alpha_q^p|u_q \quad (10.9)$$

(using the most recently defined q) until a basis change is encountered.

There can be several identical ratios. They represent the same break point (which is called a *multiple break point*) on the graph but they contribute to the decrease of the slope as if they were distinct. The multiplicity of a break point is the number of identical ratios defining it. If the maximum of $Z(t)$ is attained at a multiple break point then dual degeneracy is generated after the iteration.

The change of the dual objective function can easily be traced. First, the absolute values of the ratios need to be sorted in ascending order, $0 \leq |t_1| \leq \dots \leq |t_Q|$, where Q is the number of break points defined. Let Z_k denote the value of the dual objective at break point k and $Z_0 = \mathbf{b}^T \mathbf{y}$ (the value at the beginning of the iteration). Defining $t_0 = 0$ and following (2.68), Z_k can be computed as:

$$Z_k = Z_{k-1} + (t_k - t_{k-1})s_{k-1}, \quad k = 1, \dots, Q.$$

10.2.4 Extended General Dual algorithm

Based on the observations of the previous section now we can define an Extended General Dual algorithm for solving an LP problem in CF-1 with all types of variables present. This algorithm will be referred to as *Dual-GX*.

10.2.4.1 Step-by-step description of Dual-GX

Assumptions of Dual-GX: A basis \mathcal{B} and the index set \mathcal{U} of nonbasic variables at upper bound make a dual feasible basis to the problem. The primal solution is $\boldsymbol{\beta} = \mathbf{B}^{-1} \left(\mathbf{b} - \sum_{j \in \mathcal{U}} u_j \mathbf{a}_j \right)$.

Step 1. If $\boldsymbol{\beta}$ is primal feasible, i.e., satisfies (1.20) then it is optimal.

Otherwise, select an infeasible basic variable and denote its basic position by p (pivot row).

Step 2. Determine the nonbasic components of the updated pivot row:
 $\alpha_{\mathcal{R}}^p = \mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{R}$.

Step 3. Determine the break points of $Z(t)$ by computing dual ratios for eligible positions (t_i -a for the \mathcal{J} as defined in Step 3 of Dual-G). Store their absolute values in a sorted order: $0 \leq |t_1| \leq \dots \leq |t_Q|$.

If $Q = 0$ the problem is dual unbounded (primal infeasible), stop.

Otherwise set $k = 1$, $T^+ = T^- = \emptyset$ and s_0 according to (10.8).

Step 4. Let q denote the subscript of the variable defining t_k . t_k is the current θ_P .

If $|\theta_P| \leq u_q$ then entering variable (x_q) is found, go to Step 5. (Clearly, if type of x_q is 2 or 3 then x_q is the incoming variable because the slope in (10.9) becomes $-\infty$ as $u_q = \infty$.)

Otherwise, a bound flip is triggered. In accordance with its direction, set

$$x_q = u_q \text{ or } x_q = 0,$$

$$T^+ := T^+ \cup \{q\} \text{ or } T^- := T^- \cup \{q\}.$$

Compute $Z_k = Z_{k-1} + (t_k - t_{k-1})s_{k-1}$ and $s_k = s_{k-1} - |\alpha_q^p|u_q$. If $s_k > 0$ and $k < Q$ increment k and go to the beginning of Step 4.

If $k = Q$ and s_k is still positive then dual is unbounded, stop.

Step 5. Update solution:

1 Take care of bound flips:

$$\boldsymbol{\beta} := \boldsymbol{\beta} - \sum_{j \in T^+} u_j \boldsymbol{\alpha}_j + \sum_{j \in T^-} u_j \boldsymbol{\alpha}_j, \quad (10.10)$$

where $\boldsymbol{\alpha}_j = \mathbf{B}^{-1} \mathbf{a}_j$ and the sum is defined to be $\mathbf{0}$ if the corresponding index set is empty. Also, adjust \mathcal{U} to reflect bound flips: $\mathcal{U} := \mathcal{U} \cup T^+ \setminus T^-$.

2 Take care of basis change:

If x_{k_p} leaves at upper bound set $x_{k_p} = u_{k_p}$, otherwise, set $x_{k_p} = 0$.

Update primal and dual solutions (dual logicals) as described in Step 5 of Dual-U. Also, update \mathcal{B} and \mathbf{B}^{-1} or the LU factorization.

Verbally, Dual-GX can be interpreted in the following way: If not the smallest ratio is selected for pivoting (bound flip in Dual-GX) then the d_j 's of the bypassed small ratios change sign after the iteration. It can be compensated by putting the bypassed variables to their opposite bounds and thus maintaining the dual feasibility of the solution.

10.2.4.2 Work per iteration

It can easily be seen that the extra work required for Dual-GX is generally small.

- 1 Ratio test: the same as in Dual-G.
- 2 The break points of the piecewise linear dual objective function have to be stored and sorted. This requires extra memory for the storage, and extra work for the sorting. However, the t_k values have to be sorted only up to the point when a basis change is defined. Therefore, if an appropriate priority queue is set up for these values the extra work can be reduced sharply.
- 3 Taking care of bound flip according to (10.10) would require multiple update (FTRAN) involving all participating columns. However, the same can be achieved by only one extra FTRAN because

$$\begin{aligned}\bar{\beta} &= \beta - \sum_{j \in T^+} u_j \alpha_j + \sum_{j \in T^-} u_j \alpha_j \\ &= \beta - \mathbf{B}^{-1} \left(\sum_{j \in T^+} u_j \mathbf{a}_j - \sum_{j \in T^-} u_j \mathbf{a}_j \right) \\ &= \beta - \mathbf{B}^{-1} \tilde{\mathbf{a}}\end{aligned}$$

with the obvious interpretation of $\tilde{\mathbf{a}}$.

10.2.4.3 Degeneracy

In case of dual degeneracy, we have $d_j = 0$ for one or more nonbasic variables. If these positions take part in the ratio test they define 0 ratios. This leads to a (multiple) break point at $t = 0$. Choosing one of them results in a non-improving iteration. Assume the multiplicity of 0 is ℓ , i.e.,

$$0 = |t_1| = \dots = |t_\ell| < |t_{\ell+1}| \leq \dots \leq |t_Q| \quad (10.11)$$

Let $\alpha_{j_1}, \dots, \alpha_{j_\ell}, \dots, \alpha_{j_Q}$ denote the corresponding coefficients in the updated pivot row (omitting the subscript p , for simplicity). The maximizing break point is defined by subscript k such that $s_k > 0$ and $s_{k+1} \leq 0$, i.e.,

$$s_k = s_0 - \sum_{i=1}^k |\alpha_{j_i}| > 0 \quad \text{and} \quad s_{k+1} = s_0 - \sum_{i=1}^{k+1} |\alpha_{j_i}| \leq 0. \quad (10.12)$$

If for this k relation $k > \ell$ holds then, by (10.11), we have $|t_k| > 0$. Hence, despite the presence of degeneracy, a positive step can be made towards optimality. If $k \leq \ell$ then the step will be degenerate.

Note, s_0 is the amount of bound violation of the selected primal basic variable. Therefore, (10.12) means that if the bound violation is greater than the sum of the $|\alpha_{j_i}|$ values in the participating degenerate positions then, despite degeneracy, the algorithm makes a positive step.

10.2.4.4 Implementation

For any dual simplex algorithm based on the revised simplex method it is important to be able to access the original sparse matrix \mathbf{A} both rowwise and columnwise. Therefore, for fast access, \mathbf{A} should be stored in both ways.

For the efficient implementation of Dual-GX a sophisticated data structure (e.g., a priority queue) is needed to store and (partially) sort the break points. This part of the implementation must be done with great care because experience shows a rather erratic behavior of the algorithm regarding the generated and used break points. For example, it can happen that several thousands of break points are defined in consecutive iterations but in one of them only less than 10 are used while in the next one nearly all. Even the number of generated break points can change by orders of magnitude up and down from one iteration to the next. The tendency is that along the simplex tail these numbers go down to moderate values and often the first break point is chosen which corresponds to the step of the traditional dual method.

10.2.4.5 Features

Dual-GX possesses several advantageous features that make it the algorithm of choice for problems with upper bounded variables.

- 1 The more type-1 variables are present the more effective the method is. Namely, in such cases the chances of generating many break points is high which can lead to longer steps towards dual optimality.
- 2 Dual-GX can cope with dual degeneracy more efficiently than the traditional method as it can bypass zero valued break points and thus make a positive step.
- 3 It is also effective in avoiding the generation of dual degeneracy. If the first break point has multiplicity greater than one the traditional method will create degeneracy. Dual-GX can bypass this situation and also later break points with multiplicity (but, of course, can run into multiple break points later).
- 4 Dual-GX has a better numerical behavior due to the increased flexibility in choosing the pivot element. Namely, if the maximum of $Z(t)$

is defined by a small pivot element and this is not the first break point then we can take the neighboring (second neighbor, etc.) break point that is defined by an acceptable pivot.

- 5 In Dual-GX, the incoming variable enters the basis at feasible level.
- 6 The computational viability of Dual-GX lies in the multiple use of the expensively computed pivot row.

In short, the main advantage of the Dual-GX algorithm is that it can make many iterations with one updated pivot row and these iterations cost hardly more than one traditional iteration. Such situations frequently occur when LP relaxations of MILP problems are solved.

On the other hand, Dual-GX incorporates the usual dual algorithm and makes steps according to it when bound flip cannot be made.

10.2.4.6 Two examples for the algorithmic step of Dual-GX

The operation of Dual-GX is demonstrated on two examples. We assume the nonbasic variables are located in the leftmost positions and the nonbasic components of the updated pivot row have been determined. The types of the variables and the d_j values are given. The status of upper bounded variables is also indicated (LB for ‘at lower bound’ and UB for ‘at upper bound’).

EXAMPLE 10.1 *The problem has 10 nonbasic variables. Assume a type-2 basic variable $\beta_p = -11$ has been chosen to leave the basis, $Z_0 = 1$. Note, the solution is dual degenerate.*

j	1	2	3	4	5	6	7	8	9	10
$Type(x_j)$	0	1	1	1	1	1	1	1	1	2
$Status$	LB	LB	UB	UB	LB	UB	UB	LB		
$Upper bound$	0	1	1	1	1	1	2	1	5	∞
α_j^p	2	-2	1	3	-4	-1	1	-2	-1	-2
d_j	-1	2	5	-6	-2	0	0	0	4	10
$Ratio$		-1		-2		0	0		-4	-5

6 ratios have been defined, $Q = 6$. After sorting the absolute values of the ratios:

k	1	2	3	4	5	6
j_k	6	7	2	4	9	10
$ t_k $	0	0	1	2	4	5
$\alpha_{j_k}^p$	-1	1	-2	3	-1	-2

According to the definition of the slope, $s_0 = -\beta_p = 11$. Now, applying Step 4 of Dual-GX, we obtain

k	j_k	$ t_k $	$\alpha_{j_k}^p$	u_{j_k}	θ_P	$s_k = s_{k-1} - \alpha_{j_k}^p u_{j_k}$	$\beta_p^{(k)}$	Z_k	Note
1	6	0	-1	1	$\frac{-11}{-1} = 11$	$11 - -1 \times 1 = 10$	-10	1	$\uparrow x_6$
2	7	0	1	2	$\frac{-10}{1} = -10$	$10 - -1 \times 2 = 8$	-8	1	$\downarrow x_7$
3	2	1	-2	1	$\frac{-8}{-2} = 4$	$8 - -2 \times 1 = 6$	-6	9	$\uparrow x_2$
4	4	2	3	1	$\frac{-6}{3} = -2$	$6 - 3 \times 1 = 3$	-3	15	$\downarrow x_4$
5	9	4	-1	5	$\frac{-3}{-1} = 3$	$3 - -1 \times 5 = -2$	0	21	

Here, $\beta_p^{(k)}$ denotes the value of the selected basic variable after k sections of $Z(t)$ have been passed, Z_k is the value of the dual objective in break point k , \uparrow indicates a bound flip from lower to upper bound, while \downarrow denotes the opposite. In the first four steps the displacement of the actual incoming variable (θ_P) was always greater than its own upper bound (u_{j_k}), therefore they all signalled a bound flip and the algorithm proceeded. At the fifth break point the opposite happened, therefore x_9 (that defined t_5) became the incoming variable with a value of 3 (which is less than $u_9 = 5$). The dual steplength is 4 ($= -t_5$). At the end, we used five break points out of six resulting in a progress of the dual objective from 1 to 21. We have bypassed two degenerate vertices. Dual-U would have stopped at the first break point resulting in a non-improving (degenerate) iteration and a newly introduced infeasible basic variable $x_6 = 11 > 1 = u_6$.

It is worth looking at the situation after the Dual-GX iteration.

j	1	2	3	4	5	6	7	8	9	10
Type(x_j)	0	1	1	1	1	1	1	1	1	2
Status	UB^*	LB	LB^*	UB	UB^*	LB^*	UB	B^*		
\bar{d}_j	7	-6	9	6	-18	-4	4	-8	0	2

Here, '*' denotes a changed status in the Status row, B denotes basic. Clearly, the solution has changed quite substantially, even dual degeneracy has disappeared.

The next example shows that using up all ratios can occur easily.

EXAMPLE 10.2 This problem has 6 nonbasic variables. Now, a type-1 basic variable, $\beta_p = 14$, has been chosen to leave the basis because it is above its upper bound of 2, $Z_0 = 6$. Again, the solution is dual

degenerate.

j	1	2	3	4	5	6
$Type(x_j)$	1	1	1	1	1	2
$Status$	LB	LB	UB	UB	LB	
$Upper bound$	5	1	1	2	2	∞
α_j^p	-2	3	-2	-1	1	3
d_j	2	3	0	-2	0	9
$Ratio$	1	0	2	0	3	

5 ratios have been defined, $Q = 5$. The ratios are positive now. For uniformity, we still refer to their absolute value. After sorting them:

k	1	2	3	4	5
j_k	3	5	2	4	6
$ t_k $	0	0	1	2	3
$\alpha_{j_k}^p$	-2	1	3	-1	3

Now the slope is defined as $s_0 = \beta_p - v_p = 14 - 2 = 12$. Applying Step 4 of Dual-GX, we obtain

k	j_k	$ t_k $	$\alpha_{j_k}^p$	u_{j_k}	θ_P	$s_k = s_{k-1} - \alpha_{j_k}^p u_{j_k}$	Z_k	Note
1	3	0	-2	1	$12/(-2) = -6$	$12 - -2 \times 1 = 10$	6	$\downarrow x_3$
2	5	0	1	2	$10/(1) = 10$	$10 - 1 \times 2 = 8$	6	$\uparrow x_5$
3	2	1	3	1	$8/3$	$8 - 3 \times 1 = 5$	14	$\uparrow x_2$
4	4	2	-1	2	$5/(-1) = -5$	$5 - -1 \times 2 = 3$	19	$\downarrow x_4$
5	6	3	2	∞	$3/(3) = 1$	$-\infty$	22	

As expected the Dual-GX iteration has made a major reorganization of the solution. The following table shows the situation after the Dual-GX iteration has been completed. Dual degeneracy has, again, disappeared. For notations, see Example 10.1.

j	1	2	3	4	5	6
$Type(x_j)$	1	1	1	1	1	2
$Status$	LB	UB^*	LB^*	LB^*	UB^*	B^*
\bar{d}_j	8	-6	6	1	-3	0

From the above examples we can anticipate that in case of large scale problems with many upper bounded variables the advantages of Dual-GX have good chances to materialize.

The examples clearly demonstrate the main features of Dual-GX. Summarizing them, we can say that Dual-GX handles all types of variables efficiently. It is based on the piecewise linear nature of the dual objective if the latter is defined as a function of relaxing one basic dual equation. The distinguishing features of Dual-GX are: (i) in one iteration it can make as much progress as several iterations of the traditional dual method, (ii) using proper data structures it can be implemented efficiently so that an iteration requires hardly more work than an iteration of the traditional dual, (iii) it has inherently better numerical stability because it can create a large flexibility in finding a pivot element, (iv) it can be very effective in coping with degeneracy as it can bypass dual degenerate vertices more easily than the traditional pivot procedures.

10.3. Improving a dual infeasible solution

The dual algorithms of section 10.2 assume that the $(\mathcal{B}, \mathcal{U})$ pair defines a dual feasible solution. In this sense they are dual phase-2 algorithms. The obvious question is what can be done if such a solution is not available? As expected, dual phase-1 algorithms can be devised to find a dual feasible solution or to prove that the dual problem is infeasible. In this section a general dual phase-1 algorithm is presented for the CF-1 version of LP problems. It is based on Maros's paper [Maros, 2002c]. If it is combined with a dual phase-2 algorithm we obtain a full blown dual simplex method that can be used as an alternative to the primal simplex.

As all primal constraints are equalities the dual variables \mathbf{y} are unrestricted in sign, therefore, they are always feasible. The dual logicals are subject to the constraints spelt out in (10.2).

Since the d_j of a type-0 variable is always feasible such variables can be, and in fact are, ignored in dual phase-1. Any d_j that falls outside the feasibility range is dual infeasible. We define two index sets of dual infeasible variables:

$$\mathcal{M} = \{j : x_j = 0 \text{ and } d_j < 0\},$$

and

$$\mathcal{P} = \{j : (x_j = u_j \text{ and } d_j > 0) \text{ or } (\text{type}(x_j) = 3 \text{ and } d_j > 0)\}.$$

There is an easy way of making the d_j of upper bounded variables feasible. They can be infeasible in two different ways. Accordingly, we define two index sets:

$$\mathcal{T}^+ = \{j : \text{type}(x_j) = 1 \text{ and } j \in \mathcal{P}\}$$

$$\mathcal{T}^- = \{j : \text{type}(x_j) = 1 \text{ and } j \in \mathcal{M}\}$$

If we perform a bound flip for all such variables, the corresponding d_j -s become feasible. In this case the basis remains unchanged but the solution has to be updated:

$$\beta := \beta - \sum_{j \in T^+} u_j \alpha_j + \sum_{j \in T^-} u_j \alpha_j, \quad (10.13)$$

where $\alpha_j = \mathbf{B}^{-1} \mathbf{a}_j$, ' $:=$ ' denotes assignment, and a sum is defined to be 0 if the corresponding index set is empty. Determining α_j for all j in (10.13) would be computationally expensive. However, this entire operation can be performed in one single step for all variables involved in the following way.

$$\begin{aligned} \beta &:= \beta - \sum_{j \in T^+} u_j \alpha_j + \sum_{j \in T^-} u_j \alpha_j \\ &= \beta - \mathbf{B}^{-1} \left(\sum_{j \in T^+} u_j \mathbf{a}_j - \sum_{j \in T^-} u_j \mathbf{a}_j \right) \\ &= \beta - \mathbf{B}^{-1} \tilde{\mathbf{a}} \end{aligned}$$

with the obvious interpretation of $\tilde{\mathbf{a}}$. Having constructed $\tilde{\mathbf{a}}$, we need only one FTRAN operation with the inverse of the basis.

Assuming that such a *dual feasibility correction* has been carried out the definition of \mathcal{P} simplifies to:

$$\mathcal{P} = \{j : \text{type}(x_j) = 3 \text{ and } d_j > 0\}, \quad (10.14)$$

While this redefinition is not really necessary at this stage it is useful for later purposes.

If all variables are of type-1 any basis can be made dual feasible by feasibility correction.

The straightforward way of making the dual logicals of type-1 variables feasible suggests that their feasibility can be disregarded during dual phase-1. As type-0 variables do not play any role at all, only type-2 and -3 variables need to be considered. If the corresponding dual logicals all have become feasible a single feasibility correction step can make the logicals of the type-1 variables feasible.

As a consequence, set \mathcal{M} is redefined to be

$$\mathcal{M} = \{j : d_j < 0 \text{ and } \text{type}(x_j) \geq 2\}. \quad (10.15)$$

Using infeasibility sets of (10.14) and (10.15), the sum of dual infeasibilities can be defined as

$$f = \sum_{j \in \mathcal{M}} d_j - \sum_{j \in \mathcal{P}} d_j, \quad (10.16)$$

where any of the sums is zero if the corresponding index set is empty. It is always true that $f \leq 0$. In dual phase-1 the objective is to maximize f subject to the dual feasibility constraints. When $f = 0$ is reached the solution becomes dual feasible (subject to feasibility correction). If it cannot be achieved the dual is infeasible.

Assume row p has been selected somehow (i.e., the p -th basic variable β_p is leaving the basis). The elimination step of the simplex transformation subtracts some multiple of row p from $\mathbf{d}_{\mathcal{R}}$. If this multiplier is denoted by t then the transformed value of each d_j can be written as a function of t :

$$d_j^{(p)}(t) = d_j - t\alpha_j^p. \quad (10.17)$$

With this notation, $d_j^{(p)}(0) = d_j$ and the sum of infeasibilities as a function of t can be expressed (assuming t is small enough such that \mathcal{M} and \mathcal{P} remain unchanged) as:

$$f^{(p)}(t) = \sum_{j \in \mathcal{M}} d_j^{(p)}(t) - \sum_{j \in \mathcal{P}} d_j^{(p)}(t) = f^{(p)}(0) - t \left(\sum_{j \in \mathcal{M}} \alpha_j^p - \sum_{j \in \mathcal{P}} \alpha_j^p \right).$$

Clearly, f of (10.16) can be obtained as $f = f^{(p)}(0)$. To simplify notations, we drop the superscript from both $d_j^{(p)}(t)$ and $f^{(p)}(t)$ and will use $d_j(t)$ and $f(t)$ instead.

The change in the sum of dual infeasibilities, if t moves away from 0, is:

$$\Delta f = f(t) - f(0) = -t \left(\sum_{j \in \mathcal{M}} \alpha_j^p - \sum_{j \in \mathcal{P}} \alpha_j^p \right). \quad (10.18)$$

Introducing notation

$$v_p = \sum_{j \in \mathcal{M}} \alpha_j^p - \sum_{j \in \mathcal{P}} \alpha_j^p \quad (10.19)$$

(10.18) can be written as $\Delta f = -tv_p$. Therefore, requesting an improvement in the sum of dual infeasibilities ($\Delta f > 0$) is equivalent to requesting

$$-tv_p > 0 \quad (10.20)$$

which can be achieved in two ways:

$$\text{If } v_p > 0 \text{ then } t < 0 \text{ must hold,} \quad (10.21)$$

$$\text{if } v_p < 0 \text{ then } t > 0 \text{ must hold.} \quad (10.22)$$

As long as there is a $v_i \neq 0$ with $\text{type}(\beta_i) \neq 3$ (type-3 variables are not candidates to leave the basis) there is a chance to improve the dual objective function. The precise conditions will be worked out in the sequel. From among the candidates we can select v_p using some simple or sophisticated rule (dual phase-1 pricing).

Let the original index of the p -th basic variable β_p be denoted by $\mu (\equiv k_p)$, i.e., $x_\mu = \beta_p$ (which is selected to leave the basis). At this point we stipulate that after the basis change d_μ of the outgoing variable take a feasible value. It corresponds to releasing the p -th dual basic (equality) constraint so that it remains a satisfied inequality. This is not necessary but it gives a better control of dual infeasibilities.

If t moves away from zero (increasing or decreasing as needed) some of the d_j s move toward zero (the boundary of their feasibility domain) either from the feasible or infeasible side and at a specific value of t they reach it. Such values of t are determined by:

$$t_j = \frac{d_j}{\alpha_j^p}, \text{ for some nonbasic } j \text{ indices,}$$

and they enable a basis change since $d_j(t)$ becomes zero at this value of t , see (10.17). It also means that the j -th dual constraint becomes tight at this point. Let us assume the incoming variable x_q has been selected. Currently, d_μ of the outgoing basic variable is zero. After the basis change its new value is determined by the transformation formula of the simplex method giving

$$\bar{d}_\mu = -\frac{d_q}{\alpha_q^p} = -t_q,$$

which we want to be dual feasible. The proper sign of \bar{d}_μ is determined by the way the outgoing variable leaves the basis. This immediately gives rules how an incoming variable can be determined once an outgoing variable (pivot row) has been chosen. Below is a verbal description of these rules. Details follow in the next section.

- 1 If $v_p > 0$ then $t_q < 0$ is needed for (10.21) which implies that the p -th basic variable must leave the basis at lower bound (because \bar{d}_μ must be nonnegative for feasibility). In the absence of dual degeneracy this means that d_q and α_q^p must be of the opposite sign. In other words, the potential pivot positions in the selected row are those that satisfy this requirement.
- 2 If $v_p < 0$ then $t_q > 0$ is needed which is only possible if the outgoing variable β_p (alias x_μ) is of type-1 leaving at upper bound. In the

absence of degeneracy this means that d_q and α_q^p must be of the same sign.

- 3 If $v_p \neq 0$ and the outgoing variable is of type-0 then the sign of d_q is immaterial. Therefore, if $v_p > 0$ we look for $t_q < 0$ and if $v_p < 0$ we choose from the positive t values.

It remains to see how vector $\mathbf{v} = [v_1, \dots, v_m]^T$ which will be referred to as the *dual phase-1 reduced costs* can be computed for the row selection. In vector form, (10.19) can be written as

$$\mathbf{v} = \sum_{j \in M} \boldsymbol{\alpha}_j - \sum_{j \in P} \boldsymbol{\alpha}_j = \mathbf{B}^{-1} \left(\sum_{j \in M} \mathbf{a}_j - \sum_{j \in P} \mathbf{a}_j \right) = \mathbf{B}^{-1} \tilde{\mathbf{a}} \quad (10.23)$$

with the obvious meaning of the auxiliary vector $\tilde{\mathbf{a}}$. This operation involves an FTRAN of $\tilde{\mathbf{a}}$ which is not necessarily an expensive operation in terms of the revised simplex method.

10.3.1 Analysis of the dual infeasibility function

We can investigate how the sum of dual infeasibilities, $f(t)$, changes as t moves away from 0 ($t \geq 0$ or $t \leq 0$). It will be shown that, in either case, $f(t)$ is a piecewise linear concave function with break points corresponding to different choices of the entering variable. The global maximum of this function is achieved when its slope changes sign. It gives the maximum improvement in the sum of dual infeasibilities that can be achieved with the selected outgoing variable by making multiple use of the updated pivot row.

Let the index of the pivot row be denoted by p , the outgoing variable by β_p ($\equiv x_\mu$) and the index of the incoming variable by q . The pivot element is α_q^p .

After the basis change, the new values of d_j are determined by:

$$\bar{d}_j = d_j - \frac{d_q}{\alpha_q^p} \alpha_j^p \quad \text{for } j \in R, \quad (10.24)$$

and for the leaving variable:

$$\bar{d}_\mu = -\frac{d_q}{\alpha_q^p}.$$

The feasibility status of a d_j may change as t moves away from zero. The following analysis uses the notationally simplified form of (10.17) (i.e., $d_j(t) = d_j - t\alpha_j^p$), (10.20) and (10.24) to keep track of the changes of the feasibility status of each $d_j(t)$.

I. $t \geq 0$, i.e., the outgoing variable leaves at upper bound.

1 $\alpha_j^P > 0$, $d_j(t)$ is decreasing

(a) $d_j(0) > 0$

(i) If $d_j(0)$ is infeasible, i.e., $j \in \mathcal{P}$, it remains so as long as $t < \frac{d_j(0)}{\alpha_j^P}$ and it becomes infeasible again if $t > \frac{d_j(0)}{\alpha_j^P}$ when j joins \mathcal{M} .

(ii) If $d_j(0)$ is feasible, $d_j(t)$ remains feasible as long as $t \leq \frac{d_j(0)}{\alpha_j^P}$ after which it becomes negative and j joins \mathcal{M} .

(b) If $d_j(0) = 0$ it remains feasible only if $t = \frac{d_j(0)}{\alpha_j^P} = 0$. For $t > 0$ it becomes infeasible and j joins \mathcal{M} .

2 $\alpha_j^P < 0$, $d_j(t)$ is increasing

(a) If $d_j(0) < 0$, i.e., $j \in \mathcal{M}$, $d_j(t)$ remains infeasible as long as $t < \frac{d_j(0)}{\alpha_j^P}$. If type(x_j) = 3, it becomes infeasible again when $t > \frac{d_j(0)}{\alpha_j^P}$ and j joins \mathcal{P} .

(b) If $d_j(0) = 0$ and type(x_j) = 3 then it remains feasible only for $t = \frac{d_j(0)}{\alpha_j^P} = 0$; for $t > 0$ it becomes positive and j joins \mathcal{P} .

II. $t \leq 0$, i.e., the outgoing variable leaves at zero.

1 $\alpha_j^P > 0$, i.e., $d_j(t)$ is increasing

(a) If $d_j(0) < 0$, i.e., $j \in \mathcal{M}$, $d_j(t)$ remains infeasible as long as $t > \frac{d_j(0)}{\alpha_j^P}$. If type(x_j) = 3, it becomes infeasible again when $t < \frac{d_j(0)}{\alpha_j^P}$ and j joins \mathcal{P} .

(b) If $d_j(0) = 0$ and type(x_j) = 3 then $d_j(t)$ remains feasible only for $t = \frac{d_j(0)}{\alpha_j^P} = 0$; for $t < 0$ it becomes positive and j joins \mathcal{P} .

2 $\alpha_j^P < 0$, i.e., $d_j(t)$ is decreasing

(a) $d_j(0) > 0$ and infeasible, $j \in \mathcal{P}$, it remains so as long as $t > \frac{d_j(0)}{\alpha_j^P}$ and it becomes infeasible again if $t < \frac{d_j(0)}{\alpha_j^P}$ when j joins \mathcal{M} .

- (b) $d_j(0) = 0$, and $\text{type}(x_j) = 3$ then $d_j(t)$ remains feasible only if $t = \frac{d_j(0)}{\alpha_j^p} = 0$; for $t < 0$ it becomes negative and j joins \mathcal{M} .

The above discussion can be summarized as follows.

- 1 If $t \geq 0$ is required then the dual feasibility status of d_j (and set \mathcal{M} or \mathcal{P} , thus the composition of $f(t)$) changes for values of t defined by positions where
 $d_j < 0$ and $\alpha_j^p < 0$ or
 $d_j \geq 0$ and $\alpha_j^p > 0$
- 2 If $t \leq 0$ is required then the critical values are defined by
 $d_j < 0$ and $\alpha_j^p > 0$ or
 $d_j \geq 0$ and $\alpha_j^p < 0$.

The second case can directly be obtained from the first one by using $-\alpha_j^p$ in place of α_j^p . In both cases there are some further possibilities. Namely, if $\text{type}(x_j) = 3$ (free variable) and $d_j \neq 0$ then at the critical point the feasibility status of d_j changes twice. First when it becomes zero (feasible), and second, when it becomes nonzero again. Both cases define identical values of d_j/α_{pj} for t .

Let the critical values defined above for $t \geq 0$ be arranged in an ascending order: $0 \leq t_1 \leq \dots \leq t_Q$, where Q denotes the total number of them. For $t \leq 0$ we make a reverse ordering: $t_Q \leq \dots \leq t_1 \leq 0$, or equivalently, $0 \leq -t_1 \leq \dots \leq -t_Q$. Now we are ready to investigate how $f(t)$ characterizes the change of dual infeasibility.

Clearly, Q cannot be zero, i.e., if row p has been selected as a candidate it defines at least one critical value, see (10.19). Assuming $v_p < 0$ the initial slope of $f(t)$, according to (10.18), is

$$s_p^0 = -v_p = \sum_{j \in \mathcal{P}} \alpha_j^p - \sum_{j \in \mathcal{M}} \alpha_j^p. \quad (10.25)$$

Now $t \geq 0$ is required, so we try to move away from $t = 0$ in the positive direction. $f(t)$ keeps improving at the rate of s_p^0 until t_1 . At this point $d_{j_1}(t_1) = 0$, j_1 denoting the position that defined the smallest ratio $t_1 = d_{j_1}(0)/\alpha_{j_1}^p$. At t_1 the feasibility status of d_{j_1} changes. Either it becomes feasible at this point or it becomes infeasible after t_1 .

If $t_1 \geq 0$ then either (a) $d_{j_1} \geq 0$ and $\alpha_{j_1}^p > 0$ or (b) $d_{j_1} \leq 0$ and $\alpha_{j_1}^p < 0$. In these cases:

- (a) $d_{j_1}(t)$ is decreasing.

- (i) If d_{j_1} was feasible it becomes infeasible and j_1 joins \mathcal{M} . At his point s_p^0 decreases by $\alpha_{j_1}^p$, see (10.25).
- (ii) If d_{j_1} was infeasible ($j_1 \in \mathcal{P}$) it becomes feasible and j_1 leaves \mathcal{P} . Consequently, s_p^0 decreases by $\alpha_{j_1}^p$.

If $d_{j_1} = 0$ then we only have (i).

- (b) $d_{j_1}(t)$ is increasing.

- (i) If d_{j_1} was feasible it becomes infeasible and j_1 joins \mathcal{P} . At his point s_p^0 decreases by $-\alpha_{j_1}^p$, see (10.25).
- (ii) If d_{j_1} was infeasible ($j_1 \in \mathcal{M}$) it becomes feasible and j_1 leaves \mathcal{M} . Consequently, s_p^0 decreases by $-\alpha_{j_1}^p$.

If $d_{j_1} = 0$ then we only have (i).

Cases (a) and (b) can be summarized by saying that at t_1 the slope of $f(t)$ decreases by $|\alpha_{j_1}^p|$ giving $s_p^1 = s_p^0 - |\alpha_{j_1}^p|$. If s_p^1 is still positive we carry on with the next point (t_2), and so on. The above analysis is valid at each point. Clearly, $f(t)$ is linear between two neighboring threshold values. For obvious reasons, these values are called *break points*. The distance between two points can be zero if a break point has a multiplicity > 1 . Since the slope decreases at break points $f(t)$ is a *piecewise linear concave function* as illustrated in Figure 10.2. It achieves its maximum when the slope changes sign. This is a global maximum. If we further increase the value of t after this point the dual objective function starts deteriorating.

If $v_p > 0$ then $t \leq 0$ is required. In this case the above analysis remains valid if α_j^p is substituted by $-\alpha_j^p$. It is easy to see that both cases are covered if we take $s_p^0 = |v_p|$ and

$$s_p^k = s_p^{k-1} - |\alpha_{j_k}^p|, \text{ for } k = 1, \dots, Q.$$

10.3.2 Dual phase-1 iteration with all types of variables

Let $t_0 = 0$ and $f_i = f(t_i)$. Obviously, the sum of dual infeasibilities in the break points can be computed recursively as $f_k = f_{k-1} + s_p^{k-1}(t_k - t_{k-1})$, for $k = 1, \dots, Q$.

Below, we give the description of one iteration of the algorithm which will be referred to as GDPO (which stands for Generalized Dual Phase One).

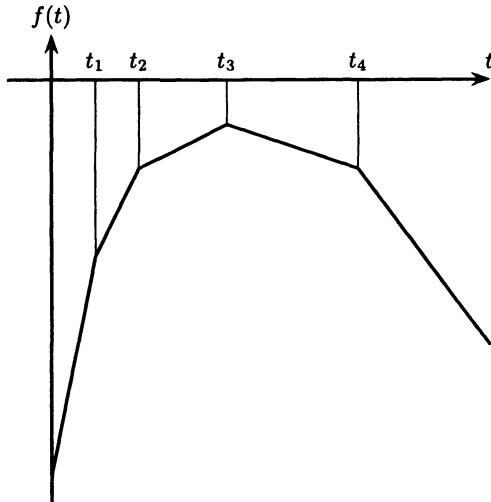


Figure 10.2. The sum of dual infeasibilities as a function of t .

An iteration of the Generalized Dual Phase-1 (GDPO) algorithm.

Step 1. Identify sets \mathcal{M} and \mathcal{P} as defined in (10.15) and (10.14). If both are empty, perform feasibility correction. After that the solution is dual feasible, algorithm terminates.

Step 2. Form auxiliary vector $\tilde{\mathbf{a}} = \sum_{j \in \mathcal{M}} \mathbf{a}_j - \sum_{j \in \mathcal{P}} \mathbf{a}_j$.

Step 3. Compute the vector of dual phase-1 reduced costs: $\mathbf{v} = \mathbf{B}^{-1} \tilde{\mathbf{a}}$, as in (10.23).

Step 4. Dual pricing: Select an improving candidate row according to some rule (e.g., Dantzig pricing [Dantzig, 1963] or normalized pricing [Forrest and Goldfarb, 1992, Harris, 1973]), denote its basic position by p . This will be the pivot row.

If none exists, terminate: The dual problem is infeasible.

Step 5. Compute the p -th row of \mathbf{B}^{-1} : $\rho^T = \mathbf{e}_p^T \mathbf{B}^{-1}$ and determine nonbasic components of the updated pivot row by $\alpha_j^p = \rho^T \mathbf{a}_j$ for $j \in \mathcal{R}$.

Step 6. Compute dual ratios for eligible positions according to rules under I., if $v_p < 0$, or II., if $v_p > 0$, as discussed in section 10.3.1. Store their absolute values in a sorted order: $0 \leq |t_1| \leq \dots \leq |t_Q|$.

Step 7. Set $k = 0$, $t_0 = 0$, $f_0 = f(0)$, $s_p^0 = |v_p|$.

While $k < Q$ and $s_p^k \geq 0$ do

$k := k + 1$

Let j_k denote the column index of the variable that defined the k -th smallest ratio, $|t_k|$.

Compute $f_k = f_{k-1} + s_p^{k-1}(t_k - t_{k-1})$, $s_p^k = s_p^{k-1} - |\alpha_{j_k}^p|$.

end while

Let q denote the index of the last breakpoint for which the slope s_p^k was still nonnegative, $q = j_k$. The maximum of $f(t)$ is achieved at this break point. The incoming variable is x_q .

Step 8. Update solution:

If x_{k_p} leaves at upper bound (the $v_p < 0$ case) set $x_{k_p} = u_{k_p}$, otherwise (the $v_p > 0$ case) set $x_{k_p} = 0$.

Compute $\alpha_q = \mathbf{B}^{-1}\mathbf{a}_q$.

Set $\theta_P = \beta_p/\alpha_q^p$ if $v_p > 0$ or $\theta_P = (\beta_p - v_p)/\alpha_q^p$ if $v_p < 0$ and update the basic solution as in Step 5 of Dual-U.

Update \mathcal{B} (and \mathcal{U} , if necessary) to reflect the basis change.

Update the reduced costs (dual logicals) as described in Step 5 of Dual-U.

Note, the dual phase-1 reduced costs (the components of \mathbf{v}) are not updated but recomputed, see Step 3 of GDPO.

10.3.2.1 Correctness of the algorithm

It remains to see that GDPO is a correct algorithm.

First, as the dual objective function defined in (10.16) has an upper bound of 0, the solution is never unbounded.

Second, maximizing f subject to the dual feasibility constraints is a convex problem. Therefore, if no improving row can be found and the dual solution is still infeasible then the problem is dual infeasible.

Third, if there is always a positive step towards feasibility no basis can be repeated as different bases have different infeasibility values which ensures finiteness. If degeneracy is present and GDPO can make only a degenerate step then the theoretically safe (and computationally efficient) ‘ad hoc’ method by Wolfe [Wolfe, 1963] can be used as long as necessary.

10.3.2.2 Work per iteration

It is easy to see that the extra work required for GDPO is generally small.

1 Ratio test: same work as with traditional dual.

2 The break points of the piecewise linear dual objective function have to be stored and sorted. This requires extra memory for the storage, and extra work to sort. However, the t_k values have to be sorted only up to the point where $f(t)$ reaches its maximum. Therefore, if an appropriate priority queue is set up for these values the extra work can be kept at minimum.

10.3.2.3 Coping with degeneracy

In case of dual degeneracy, we have $d_j = 0$ for one or more nonbasic variables. If these positions take part in the ratio test they define 0 ratios. This leads to a (multiple) break point at $t = 0$. Choosing one of them results in a non-improving iteration. Assume the multiplicity of 0 is ℓ , i.e.,

$$0 = |t_1| = \dots = |t_\ell| < |t_{\ell+1}| \leq \dots \leq |t_Q| \quad (10.26)$$

Let $\alpha_{j_1}, \dots, \alpha_{j_\ell}, \dots, \alpha_{j_Q}$ denote the corresponding coefficients in the updated pivot row (omitting superscript p , for simplicity). The maximizing break point is defined by subscript k such that $s_k > 0$ and $s_{k+1} \leq 0$, i.e.,

$$s_k = s_0 - \sum_{i=1}^k |\alpha_{j_i}| > 0 \quad \text{and} \quad s_{k+1} = s_0 - \sum_{i=1}^{k+1} |\alpha_{j_i}| \leq 0.$$

If for this k relation $k > \ell$ holds then, by (10.26), we have $|t_k| > 0$. Hence, despite the presence of degeneracy, a positive step can be made towards dual feasibility. If $k \leq \ell$ then the step will be degenerate.

10.3.2.4 Implementation

For the efficient implementation of GDPO a sophisticated data structure (priority queue) is needed to store and (partially) sort the break points. Additionally, since row and column operations are performed on \mathbf{A} , it is important to have a data structure that supports efficient row and columnwise access of \mathbf{A} .

10.3.2.5 Two examples of the algorithmic step

The operation of GDPO is demonstrated on two examples. We assume pivot row p has been selected based on v_p and the updated pivot row has been determined. The types of the variables and the d_j values are

given. For obvious reasons, the examples do not contain type-0 and type-1 variables.

EXAMPLE 10.3 *The problem has 8 nonbasic variables. Now $v_p = 10$ which is the $t \leq 0$ case. The sum of dual infeasibilities is $f = -35$. The solution is dual degenerate.*

j	1	2	3	4	5	6	7	8
$\text{type}(x_j)$	3	3	3	2	2	2	2	2
Status								
α_j^p	8	4	1	4	2	-1	-1	1
d_j	-24	2	0	-8	-1	1	0	0
Infeasibility	M	P	M	M				
Ratio	-3	0	-2	-0.5	-1	0		
2nd ratio	-3							

Altogether, 7 ratios have been defined, so $Q = 7$. Note, for the first position ($j = 1$) there are two identical ratios. After sorting the absolute values of the ratios we obtain:

Index	1	2	3	4	5	6	7
j_k	7	3	5	6	4	1	1
$ t_{j_k} $	0	0	0.5	1	2	3	3
$\alpha_{j_k}^p$	-1	1	2	1	4	8	8

Now, we can apply Step 7 of GDPO:

k	j_k	$ t_k $	$\alpha_{j_k}^p$	$f_k = f_{k-1} + s_p^{k-1}(t_k - t_{k-1})$	$s_p^k = s_p^{k-1} - \alpha_{j_k}^p $	
0	0			-35	10	
1	7	0	-1	-35	$10 - -1 = 9$	
2	3	0	1	-35	$9 - 1 = 8$	
3	5	0.5	2	$-35 + 8 \times (0.5 - 0) = -31$	$8 - 2 = 6$	
4	6	1	1	$-31 + 6 \times (1 - 0.5) = -28$	$6 - 1 = 5$	
5	4	2	4	$-28 + 5 \times (2 - 1) = -23$	$5 - 4 = 1$	
6	1	3	8	$-23 + 1 \times (3 - 2) = -22$	$1 - 8 = -7$	

We have used 6 of the 7 breakpoints. At termination of this step of GDPO $k = 6$, therefore, the entering variable is x_1 that has defined t_6 . The dual steplength is -3 (= t_6). Traditional methods would have stopped at the first breakpoint resulting in a degenerate iteration.

It is instructive to monitor how dual infeasibility changes at the non-zero break points. In the next table we show how the d_j values (the dual logicals) would be transformed if the iteration stopped at different break points taken them in an increasing order.

j	1	2	3	4	5	6	7	8	
$\text{type}(x_j)$	3	3	3	2	2	2	2	2	
α_j^p	8	4	1	4	2	-1	-1	1	
d_j	-24	2	0	-8	-1	1	0	0	$f = -35$
<i>Infeasibility</i>	\mathcal{M}	\mathcal{P}	\mathcal{M}	\mathcal{M}					
$d_j + 0.5\alpha_j^p$	-20	4	0.5	-6	0	0.5	-0.5	0.5	$f = -31$
<i>Infeasibility</i>	\mathcal{M}	\mathcal{P}	\mathcal{P}	\mathcal{M}			\mathcal{M}		
$d_j + 1 \times \alpha_j^p$	-16	6	1	-4	1	0	-1	1	$f = -28$
<i>Infeasibility</i>	\mathcal{M}	\mathcal{P}	\mathcal{P}	\mathcal{M}			\mathcal{M}		
$d_j + 2 \times \alpha_j^p$	-8	10	2	0	3	-1	-2	2	$f = -23$
<i>Infeasibility</i>	\mathcal{M}	\mathcal{P}	\mathcal{P}		\mathcal{M}	\mathcal{M}			
$d_j + 3 \times \alpha_j^p$	0	14	3	4	5	-2	-3	4	$f = -22$
<i>Infeasibility</i>		\mathcal{P}	\mathcal{P}		\mathcal{M}	\mathcal{M}			

At the end of Step 7 of GDPO we still have four infeasible positions, $\mathcal{P} = \{2, 3\}$, $\mathcal{M} = \{6, 7\}$. Some infeasibilities disappeared and new ones were created but the sum of infeasibilities has been reduced from -35 to -22 . It is interesting to see what happens if we increase t to 4. The outcome is $f = -37$ which shows a rate of deterioration of 15. The reason for this rate is that at $t = 3$ we have a break point with multiplicity of 2 (see line '2nd ratio'), both defined by x_1 . Therefore, going beyond it, the slope decreases by $2|\alpha_1^p| = 16$ from +1 to -15.

EXAMPLE 10.4 This problem has 4 nonbasic variables. Here, $v_p = -5$ which is the $t \geq 0$ case. The sum of dual infeasibilities is $f = -11$.

j	1	2	3	4	
$\text{type}(x_j)$	2	2	2	3	
α_j^p	-2	-3	-1	2	$v_p = -4$
d_j	-4	0	-1	6	$f = -11$
<i>Infeasibility</i>	\mathcal{M}	\mathcal{M}	\mathcal{P}		
<i>Ratio</i>	2		1	3	
<i>2nd ratio</i>				3	

As 4 ratios have been defined, $Q = 4$. After sorting the ratios which are nonnegative now:

Index	1	2	3	4
j_k	3	1	4	4
t_{j_k}	1	2	3	3
$\alpha_{j_k}^p$	-1	-2	2	2

Applying Step 7 of GDPO, we obtain

k	j_k	t_k	$\alpha_{j_k}^p$	$f_k = f_{k-1} + s_p^{k-1}(t_k - t_{k-1})$	$s_p^k = s_p^{k-1} - \alpha_{j_k}^p $
1	3	1	-1	$-11 + 5 \times (1 - 0) = -6$	$5 - -1 = 4$
2	1	2	-2	$-6 + 4 \times (2 - 1) = -2$	$4 - -2 = 2$
3	4	3	2	$-2 + 2 \times (3 - 2) = 0$	$2 - 2 = 0$

Altogether 3 break points have been used. The last break point defines the dual steplength of 3 ($= t_3$) and the entering variable $x_{j_3} = x_4$. In this case GDPO reaches dual feasibility with one updated pivot row in one iteration.

Again, we can monitor how infeasibility changes with different choices of the pivot column. Below we show what happens if we stop at the first, second or the third break point.

j	1	2	3	4	
$type(x_j)$	2	2	2	3	
α_j^p	-2	-3	-1	2	
d_j	-4	0	-1	6	$f = -11$
Infeasibility	\mathcal{M}	\mathcal{M}	\mathcal{P}		

$d_j - 1 \times \alpha_j^p$	-2	3	0	4	$f = -6$
Infeasibility	\mathcal{M}		\mathcal{P}		

$d_j - 2 \times \alpha_j^p$	0	6	1	2	$f = -2$
Infeasibility				\mathcal{P}	

$d_j - 3 \times \alpha_j^p$	2	9	2	0	$f = 0$
Infeasibility					

Obviously, tracing infeasibility as shown in the examples above is actually not done in GDPO as there is no need for that. It was included only to demonstrate the outcome of different possible choices of the incoming variable.

10.3.2.6 Key features of GDPO

The main computational advantage of GDPO is that it can make multiple steps with one updated pivot row. These steps correspond to several traditional iterations. Such a multiple step of GDPO requires very little extra work compared to standard dual phase-1 techniques.

GDPO is a generalization of the traditional dual phase-1 simplex algorithm in the sense that the latter stops at the smallest ratio (first break point) while GDPO can pass many break points making the maximum progress towards dual feasibility with the selected outgoing variable. Its efficiency is not hampered by the presence of all types of variables.

GDPO possesses a potentially favorable anti-degeneracy property as shown in section 10.3.2.3.

The freedom of multiple choice created by the break points can be used to enhance the numerical stability of GDPO, in a similar fashion as for Dual-GX. Namely, if the maximum of $f(t)$ is defined by a small pivot element and this is not the first break point then we can take the neighboring break point (second neighbor, etc., as long as $f(t)$ improves) if it is defined by an acceptable pivot.

GDPO works within the framework of the simplex method. It approaches a dual feasible solution (if one exists) by basis changes. It is a sort of a greedy algorithm. This is a locally best strategy. There is no guarantee that, ultimately, it will lead to a faster termination of phase-1 than other methods. Computational experience, however, shows that in many cases it can reduce the total number of dual phase-1 iterations substantially. On the other hand, there are also examples when GDPO makes slightly more iterations than other methods. It appears that the final balance is in favor of GDPO.

In the light of the advantages of GDPO, it is the number one choice for a dual phase-1 algorithm if all types of variables are present in the CF-1 version an LP problem.

10.3.2.7 Handling type-4 variables

If there are type-4 (nonpositive) variables in the problem (CF-2) then GDPO has to be modified just slightly.

The dual feasibility for type-4 variables means $d_j \leq 0$. Therefore, the definition of the index set of the positively infeasible dual logicals \mathcal{P} in (10.14) need to be amended to become

$$\mathcal{P} = \{j : (\text{type}(x_j) = 3 \text{ or } \text{type}(x_j) = 4) \text{ and } d_j > 0\}.$$

It is easy to see from $d_j(t) = d_j - t\alpha_j^p$ that type-4 nonbasic variables define break points if

I. $t \geq 0$ is required

	$\alpha_j^p > 0$	$\alpha_j^p < 0$
$d_j > 0$	d_j/α_j^p	—
$d_j \leq 0$	—	d_j/α_j^p

II. $t \leq 0$ is required

	$\alpha_j^p > 0$	$\alpha_j^p < 0$
$d_j > 0$	—	d_j/α_j^p
$d_j \leq 0$	d_j/α_j^p	—

10.4. Row selection (dual pricing)

The importance of selecting the outgoing variable (pivot row) in the dual equals the importance of selecting the incoming variable in the primal.

In dual phase-2 the violation of the primal feasibility of the basic solution plays the role of the reduced cost in primal. If ω denotes the vector of violations its i -th component is defined in the following way

$$\omega_i = \begin{cases} -\beta_i & \text{if } \beta_i < 0, \text{ type}(\beta_i) \neq 3 \text{ (type-L),} \\ \beta_i - v_i & \text{if } \beta_i > v_i \text{ (type-U),} \\ 0 & \text{otherwise.} \end{cases} \quad (10.27)$$

Keeping it in mind, the main pricing tactics and techniques are, in principle, applicable for the row selection. However, there are some differences that need to be taken into account.

As the primal solution is always kept in an explicit form, there is no question about maintaining the dual reduced costs as it is done automatically. Therefore, full pricing is a more straightforward idea here than in the primal. On the other hand, there is no obstacle to partial pricing and sectional pricing with all variations.

The use of multiple pricing followed by suboptimization looks less useful in the dual. It would require the determination of several potential pivot rows which is usually a more expensive operation than the updating of several columns in the primal. Additionally, it is slightly more prone to numerical inaccuracies due to the performed dot products.

Interestingly, normalized pricing is simpler and requires less extra computations in the dual than in the primal. Therefore, it is highly recommended to include (some of) them in an advanced implementation of the dual.

First we overview some dual pricing methods in case the solution is dual feasible (dual phase-2). After it we point out what changes are needed for pricing in dual phase-1. Among the unnormalized methods the Dantzig pricing is presented briefly followed by two powerful normalized methods, the Devex and steepest edge procedures.

10.4.1 Dantzig pricing

The idea is to select the most violating row from among all or just a subset of all violations. The criterion simply requires the scanning of the ω_i values of (10.27) and choosing the largest one. Formally, index p of the pivot row is determined by

$$\omega_p = \max\{\omega_i\}. \quad (10.28)$$

If $\omega_p = 0$ the solution is primal feasible, therefore optimal. Otherwise, the p is the selected pivot row and x_{k_p} is the outgoing variable. If there is a tie then, in theory, any of the rows in the tie can be chosen. In practice, it is advisable to give priority to the removal of variables with small feasibility range (type-0 and type-1 basic variables).

If ω_p in (10.28) was defined by a type-L primal infeasibility then the outgoing variable will leave at lower bound otherwise, (type-U) at upper bound.

As the ω_i values are available for free the Dantzig pricing is very fast. Its simplicity and speed explain its popularity. Therefore, its inclusion in an implementation of the dual is straightforward and recommended. The quality of the selection is comparable to the quality of its counterpart in the primal. In other words, there are more effective methods but they are really more complicated.

10.4.2 Steepest edge

Similarly to the primal, the steepest edge pricing can be interpreted also in the framework of the dual. Depending on how the dual is formulated several dual steepest edge methods can be designed. In this section we discuss one of them following the footsteps of Forrest and Goldfarb. Several more exact and approximate dual methods can be found in their paper [Forrest and Goldfarb, 1992].

For simplicity, temporarily it is assumed that the LP problem is in CF-1 with no free and bounded variables, i.e.,

$$\min\{\mathbf{c}^T \mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$$

The dual of this problem is (see section 2.3)

$$\max\{\mathbf{b}^T \mathbf{y} : \mathbf{A}^T \mathbf{y} \leq \mathbf{c}\}.$$

Let \mathbf{B} be a dual feasible basis and \mathbf{A} partitioned as $\mathbf{A} = [\mathbf{B} \mid \mathbf{R}]$, i.e., the basic variables are in the first m positions now. If \mathbf{c}_B and \mathbf{c}_R denote the basic and nonbasic parts of \mathbf{c} then dual feasibility for this basis, according to (2.58) and (2.59), means that

$$\mathbf{B}^T \mathbf{y} = \mathbf{c}_B, \quad (10.29)$$

$$\mathbf{R}^T \mathbf{y} \leq \mathbf{c}_R. \quad (10.30)$$

During a basis change, first a basic equation in (10.29), say the p -th, is relaxed to $\mathbf{a}_p^T \mathbf{y} \leq c_p$. We say that the constraint is negatively relaxed because $\mathbf{a}_p^T \mathbf{y}$ is decreased. The change in the p -th basic equation is parameterized by $t \leq 0$ and the dual solutions can be written as

$$\mathbf{y}(t) = \mathbf{y} + t\boldsymbol{\rho}_p, \quad (10.31)$$

where $\mathbf{y} = \mathbf{B}^{-T} \mathbf{c}_B$ and $\boldsymbol{\rho}_p = \mathbf{B}^{-T} \mathbf{e}_p$, i.e., $\boldsymbol{\rho}_p$ is the p -th column of transpose of the basis inverse. If it is viewed as the p -th row of the basis inverse then it is denoted by $\boldsymbol{\rho}^p$ ($= \mathbf{e}_p^T \mathbf{B}^{-1}$).

As dual solutions of the form of (10.31) satisfy $\mathbf{a}_i^T \mathbf{y}(t) = c_i$ for $i = 1, \dots, m$, $i \neq p$ and $\mathbf{a}_p^T \mathbf{y} \leq c_p$ the rows of \mathbf{B}^{-1} , denoted by $\boldsymbol{\rho}^i$, $i = 1, \dots, m$, are the edge directions in the dual polyhedron $\{\mathbf{y} \in \mathbb{R}^m : \mathbf{A}^T \mathbf{y} \leq \mathbf{c}\}$ emanating from the vertex $\mathbf{y} = \mathbf{B}^{-T} \mathbf{c}_B$.

The gradient of the dual objective function is \mathbf{b} . Choosing an outgoing variable means choosing an edge direction. It must be such that the edge direction makes an obtuse angle with \mathbf{b} , formally, $\boldsymbol{\rho}_p^T \mathbf{b} = \mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{b} = x_p$ must be negative. This is just the known criterion for selecting a leaving variable, see (2.69).

If we want the edge to be the steepest then we must have

$$\frac{\mathbf{b}^T \boldsymbol{\rho}_p}{\|\boldsymbol{\rho}_p\|} = \min_{1 \leq i \leq m} \left\{ \frac{\mathbf{b}^T \boldsymbol{\rho}_i}{\|\boldsymbol{\rho}_i\|} \right\},$$

where $\|\cdot\|$ denotes the Euclidean norm. The square of the norm is defined as

$$\delta_i = \boldsymbol{\rho}_i^T \boldsymbol{\rho}_i = \|\boldsymbol{\rho}_i\|^2, \quad i = 1, \dots, m.$$

Recomputing the norms in every iteration according to the actual basis would require prohibitively much work. Fortunately, updating formulas can be developed that require considerably less work per iteration making this methodology practicable.

Once the outgoing variable is determined row p of \mathbf{B}^{-1} needs to be computed. Assume the incoming variable is x_q with $\boldsymbol{\alpha}_q = \mathbf{B}^{-1} \mathbf{a}_q$. After the basis change $\bar{\mathbf{B}}^{-1} = \mathbf{E} \mathbf{B}^{-1}$, where \mathbf{E} is the elementary transformation

matrix (2.24) formed from α_q . Therefore, the rows of $\bar{\mathbf{B}}^{-1}$ are updated as

$$\bar{\rho}^p = \frac{1}{\alpha_q^p} \rho^p, \quad (10.32)$$

$$\bar{\rho}^i = \rho^i - \frac{\alpha_q^i}{\alpha_q^p} \rho^p, \quad i = 1, \dots, m, \quad i \neq p. \quad (10.33)$$

Consequently, the norms after the basis change are

$$\bar{\delta}_p = \bar{\rho}^p \bar{\rho}_p = \left(\frac{1}{\alpha_q^p} \right)^2 \delta_p, \quad (10.34)$$

and

$$\begin{aligned} \bar{\delta}_i &= \bar{\rho}^i \bar{\rho}_i \\ &= \left(\rho^i - \frac{\alpha_q^i}{\alpha_q^p} \rho^p \right) \left(\rho_i - \frac{\alpha_q^i}{\alpha_q^p} \rho_p \right) \\ &= \delta_i - 2 \frac{\alpha_q^i}{\alpha_q^p} \rho^p \rho_i + \left(\frac{\alpha_q^i}{\alpha_q^p} \right)^2 \delta_p, \quad i = 1, \dots, m, \quad i \neq p. \end{aligned} \quad (10.35)$$

In (10.35) the computationally critical term is $\rho^p \rho_i$ as during a dual iteration the ρ_i columns of \mathbf{B}^{-T} (or, equivalently, the ρ^i rows of \mathbf{B}^{-1}) are not available. Since $\rho^p \rho_i = \rho^i \rho_p$ and $\rho^i = \mathbf{e}_i^T \mathbf{B}^{-1}$, we can write

$$\tau = \mathbf{B}^{-1} \rho_p \quad (10.36)$$

and $\mathbf{e}_i^T (\mathbf{B}^{-1} \rho_p) = \tau_i$. Finally, (10.35) can be replaced by

$$\bar{\delta}_i = \delta_i - 2 \frac{\alpha_q^i}{\alpha_q^p} \tau_i + \left(\frac{\alpha_q^i}{\alpha_q^p} \right)^2 \delta_p, \quad i = 1, \dots, m, \quad i \neq p. \quad (10.37)$$

As a result, the dual steepest edge weights can be updated exactly using (10.34) and (10.37). As ρ_p is always computed for a dual iteration, the ‘only’ extra work is determining τ by (10.36) which requires an FTRAN operation on the p -th row of the basis inverse.

Comparing the computational requirements of the primal and the presented dual steepest edge algorithms it becomes clear that the dual is ‘cheaper’. Though both require the solution of some extra set of equations (primal: $\alpha_q^T \mathbf{B}^{-1}$, dual: $\mathbf{B}^{-1} \rho_p$), in the dual no further computation is needed for the update of the δ weights while in primal an extra dot product has to be performed with all nonbasic columns for which $\alpha_j^p \neq 0$.

It can be shown (see [Forrest and Goldfarb, 1992]) that if upper bounded variables are also present the edge directions are practically the same as without them. Therefore the updating formulas (10.34) and (10.37) remain valid.

In practice, the dual steepest edge algorithms proved to be very powerful in terms of reducing the number of iterations considerably. Though the cost of an iteration is higher it is more than offset by the reduction of the total solution time in most practical cases.

Inclusion of at least one version of the steepest edge dual pricing algorithms is a must for an advanced implementation of the dual simplex.

10.4.3 Devex

In her seminal paper [Harris, 1973] Harris outlined the dual version of the Devex pricing which is nowadays viewed as an approximate steepest edge method. It shares several common features with its primal counterpart. The most important one is that the approximate weights keep growing in size and a resetting has to take place from time to time. Its distinguishing advantage is the very little extra work if the updated row of reduced costs is maintained. The method can be described in the following way.

The edge directions are the same as in the steepest edge method. The norms of the directions are replaced by approximate weights. The reference framework within which the Devex norm is defined is initially and periodically set to consist of the indices of the current basic variables which we denote by \mathcal{H} . At an arbitrary iteration the row weights are determined by those coefficients of the updated row which belong to the latest reference framework.

Denote $\mathbf{h} \in \mathbb{R}^m$ the vector of the approximate weights. At the time when a new framework is set up the h_i row weights are set to 1 for all $i = 1, \dots, m$. At subsequent iterations they are updated by the formulas

$$\bar{h}_p = \max\{1, \|\hat{\alpha}^p\|/\alpha_q^p\}, \quad (10.38)$$

$$\bar{h}_i = \max\{h_i, |\alpha_q^i/\alpha_q^p| \|\hat{\alpha}^p\|\}, \quad 1 \leq i \leq m, i \neq p, \quad (10.39)$$

where $\hat{\alpha}^p$ is a subvector of the updated pivot row containing elements belonging to the current reference framework. As the updated pivot row α^p is always computed in the dual algorithm, updating the weights according to (10.38) and (10.39) comes practically for free.

Since h_p can be computed exactly as $h_p = \|\hat{\alpha}^p\|$ by taking just the $\alpha_j^p, j \in \mathcal{H}$ components of α^p , it can be compared with its approximate value obtained through updating. If they differ relatively much (e.g., the estimated is three times the accurate value) then a resetting is triggered.

Because of its simplicity and good performance, the dual Devex pricing algorithm is an excellent candidate for inclusion in an advanced implementation of the dual simplex method.

10.4.4 Pricing in dual phase-1

The dual phase-1 reduced costs are not automatically available. They need to be recomputed in each iteration according to (10.23) as updating would be quite complicated because the dual infeasibility sets keep changing (they can reduce or even expand).

Once the dual phase-1 reduced costs are available any of the dual pricing techniques can be used. In an overwhelming majority of the cases any version of normalized pricing is substantially more effective than the unnormalized methods. Therefore, the minimum requirement is the inclusion and use of the dual Devex method. In case of a more demanding implementation at least one version of the dual steepest edge pricing schemes has to be made available.

10.5. Computing the pivot row

A critical operation in the dual simplex method is the computation of the nonbasic coefficients of the updated pivot row. If it is row p then the updated form is $\alpha^p = \mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{R}$ (recall, \mathbf{R} consists of the nonbasic columns of \mathbf{A}). In section 9.4 it was shown how the rowwise data structure of \mathbf{A} can help the fast updating of the reduced costs in the primal. The same idea can be applied here in the following way.

Assume there is enough space to store \mathbf{A} not only columnwise but also rowwise. In this view

$$\mathbf{R} = \begin{bmatrix} \mathbf{r}^1 \\ \vdots \\ \mathbf{r}^i \\ \vdots \\ \mathbf{r}^m \end{bmatrix},$$

where \mathbf{r}^i denotes the subvector of \mathbf{a}^i (row i of \mathbf{A}) containing its nonbasic components. Thus the $\mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{R}$ product can be written as

$$\alpha^p = \mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{R} = \rho^p \mathbf{R} = [\rho_1^p, \dots, \rho_i^p, \dots, \rho_m^p] \begin{bmatrix} \mathbf{r}^1 \\ \vdots \\ \mathbf{r}^i \\ \vdots \\ \mathbf{r}^m \end{bmatrix} = \sum_{i=1}^m \rho_i^p \mathbf{r}^i. \quad (10.40)$$

In other words, if $\mathbf{e}_p^T \mathbf{B}^{-1} \mathbf{R}$ is computed according to (10.40) then the only rows that are involved in the updating of the pivot row are the ones that correspond to nonzero coefficients of ρ^p . As ρ^p can be very sparse (very often just few nonzeros), this way of updating α^p can be computationally very efficient. Note, if \mathbf{A} is stored only columnwise form the sparsity of ρ^p cannot be utilized.

10.6. Degeneracy in the dual

Degeneracy can occur also in the dual and may cause similar problems as in the primal. Though the presented dual phase-1 and phase-2 procedures (see sections 10.3.2.3 and 10.2.4.3) have some inherent algorithmic anti-degeneracy properties it is not always enough.

Issues of handling degeneracy in the primal have been presented in section 9.7. Due to the large symmetry between the primal and the dual, most of the techniques of the primal can be applied to cope with dual degeneracy. For practical purposes, degeneracy and near degeneracy get the same treatment. For definition of near degeneracy, see (9.109).

Degeneracy in the dual manifests itself in zero reduced costs d_j (dual logicals) for nonbasic variables. If such a position is involved in the dual ratio test then there is a chance of a zero progress (again, see sections 10.3.2.3 and 10.2.4.3). Note, in traditional dual methods this case definitely results in no progress. Degeneracy or near degeneracy can also lead to stalling.

For numerical reasons, it is necessary to allow dual feasibility to be slightly violated. It allows finding better sized pivot elements which is absolutely vital for the stability of the algorithm. This can help, without any further action, leave a degenerate dual vertex.

While, in theory, it is possible to pay attention to degeneracy during dual pricing (see section 9.7.1 for a similar situation in the primal), in practice it is not done because it would require too much extra work. However, techniques of determining the pivot position in case of degeneracy can be implemented nearly in the same sense as in the primal.

The EXPAND procedure of section 9.7.3 can easily be interpreted in the framework of the dual. Therefore it is certainly a good candidate for inclusion.

Wolfe's 'ad hoc' method (section 9.7.2) is a degeneracy-aware technique. It requires the identification of the (near) degenerate positions. The identification can be done using the (relative or absolute) primal optimality tolerance. After this the 'ad hoc' method can be transcribed in terms of the dual in a relatively straightforward way. Details are omitted here. The author of this book has very favorable experience with and high appreciation for this technique which really excels itself

in the efficient solution of the LP relaxation of combinatorial optimization problems.

Perturbation and shifting as described for the primal (see section 9.7.4) can also be translated into the dual. In fact, shifting is considered a simple but very effective method of coping with dual degeneracy and is strongly recommended to be included in an advanced implementation of the dual simplex method.

The question ‘When to activate anti-degeneracy techniques?’ emerges also in the dual. In case of EXPAND the answer is easy as it is a degeneracy-unaware method. It works always the same way, independent of the situation. The degeneracy-aware methods require the identification of the degenerate positions and also have a set-up overhead. Therefore, it is worth waiting with the activation of such methods. Sometimes degeneracy goes away quickly by normal simplex iterations. However, if the objective value does not change for a certain number of iterations (or the change remains in the magnitude of the error tolerance) then it is advisable to activate such an algorithm. The number of ‘idle iterations’ is usually the refactorization (reinversion) frequency of the basis. When degeneracy disappears it is worth returning to a normal dual pricing.

Chapter 11

VARIOUS ISSUES

The title of this chapter could also be ‘Miscellaneous aspects of implementing the simplex method’. Here we just summarize some of the main points of the previous chapters and also indicate further issues of the simplex method, its algorithmic techniques and implementation that have not been discussed in the book due to some volume and time limitations. We believe the most important questions have been properly addressed and a reader, with a reasonable programming background, is now able to create a highly advanced implementation of the simplex method for solving large scale LP problems.

In this final chapter we briefly touch some additional issues to give further insight into requirements and possibilities of the computational versions of the simplex method. To give a complete account of everything in this topic is impossible. Therefore, the forthcoming selection is an arbitrary one and the author claims responsibility for its possible shortcomings.

11.1. Run parameters

In several places of the simplex method alternative algorithmic techniques have been presented. During the solution of a concrete problem the actually used units have to be specified. This is a sort of a qualitative decision. Additionally, there are numerical parameters that have to be assigned a value, like the number of variables in multiple pricing or the value of the drop tolerance. All the values that have to be specified are jointly called *run parameters*. They usually have default values but the user can change (nearly) any of them. The default values are set in the program and are inaccessible unless there is a tool to modify them externally.

11.1.1 Parameter file

The run parameters are best specified in a separate *parameter file*. It is usually a text file that can be created and modified by a text editor. In more user friendly systems there is an intuitive interface where the parameters are presented to the user and changes can be made safely (with undo facility) if needed.

The structure of a parameter file is implementation dependent. It is typical that the parameters can be referenced by meaningful names and their values are specified by assignment statements, like in the algebraic languages. Provision must be made for the inclusion of comments. A good practice is that there is a manufacturer-prepared parameter file delivered with the solver that lists all parameters with default values and annotations.

Though in most of the cases the default settings lead to a good performance there are situations when it is advisable or necessary to change some of them.

In the next subsections we give a qualitative overview of the parameters that are typical in simplex based optimization systems. In practice, every parameter has a default value determined by the developer of the system. Though the parameters below are presented in some logical grouping no ordering is required in the parameter file. A functional unit that interprets the parameter file is not too complicated to implement. If the parameter file is missing every parameter is taken at its default value. Even the expected MPS input file can have a name like `default.mps`.

The parameters can roughly be categorized as *string*, *algorithmic* and *numerical* parameters.

11.1.2 String parameters

The complete definition of an LP problem and its solution requires the identification of several names. In most cases they are the name of the MPS input file, the names of the actually used RHS, range and bound vectors if the MPS file contains several of them (not a common practice nowadays), the name of the file that contains an assumed (complete or incomplete) starting basis, names of different output files (iteration log, intermediate and optimal bases for later use, final solution), etc. The boundaries of the fixed sectional pricing method can also be given by the names of the corresponding variables. If the implementation is prepared to take advantage of a specific hardware feature it can also be identified by a name. Several other string parameters can be added to this list.

11.1.3 Algorithmic parameters

Usually, the intended main solution algorithm (primal or dual) is selected first. It is followed by the specification of the preprocessing modes (level of presolve, scaling method). Next, the starting procedure is identified, then the pricing strategy. The latter can have several numerical parameters, like the number of vectors in the suboptimization, the value of the multiplier of the true objective function if composite objective is used in phase-1, the level of inaccuracy that triggers a Devex resetting, and many more. Details have been discussed at the description of the relevant algorithmic units.

11.1.4 Numerical parameters

One part of the numerical parameters serves the selected algorithms, as indicated above. The sense of the optimization (min or max) is also defined here. It is worth setting an upper limit on the number of iterations. It stops an unattended run if the algorithm falls into a cycle (due to numerical troubles or otherwise). A simplex solver uses several tolerances. Changing their default values can be done here. Certain parameters are needed to control the frequency of refactorizations (which may be overridden by a supervisory algorithm if numerical difficulties warrant the recomputation of the LU) and regular saving of the intermediate bases.

11.2. Performance evaluation, profiling

Reliability is the number one requirement of any optimization system. It can be achieved by choosing algorithmic techniques with better numerical properties (e.g., LU instead of PFI), applying preprocessing and using tolerances. Once reliability is established efficiency becomes important. The ultimate goal is to reduce the total solution time. While it is easy to say, it is more difficult to see how it can be accomplished.

A natural idea is to monitor the time spent in different algorithmic units to locate the hot spots. This action is called *profiling*. Profiling is assisted by some compilers of high level programming languages. It is not standardized and it has to be found out for each specific case how it can be used, if at all. If the units with the longest time are identified one can concentrate on improving them.

Profiling various implementations of the simplex method has shown that the percentage of time spent in different units is very much dependent on the problem and the solution strategy. As an example, we present some of excerpts of observations by Maros and Mitra [Maros and Mitra, 2000]. Their investigated algorithmic units are listed in Ta-

Table 11.1. A grouping of the main algorithmic components of SSX

Component	Brief explanation
CHKFEZ	Checking feasibility of the current solution and, accordingly, constructing Phase-1 or Phase-2 from vector v .
BTRAN	Backward transformation: operation $\pi^T = v^T B^{-1}$ leading to the price vector π .
PRICE	Computation of the d_j reduced cost for a designated set of nonbasic variables.
FTRAN	Forward transformation: updating the column(s) selected by PRICE.
CHUZRO	Choosing the pivot row (determining the outgoing variable) by applying the ratio test.
HUZKEP	“House-keeping” step after each iteration whereby basis or bound changes and other related information are updated.
FACTOR	Refactorization of the current basis to refresh the <i>LU</i> representation of B^{-1} .
OTHER	Remaining computational work not accounted for above.

Table 11.2. Problem statistics of the test set. Models are from `netlib/lp/data`.

Problem	Number of		
	rows	columns	nonzeros
bnl2	2325	3489	16124
fit2p	3001	13525	60784
maros-r7	3137	9408	151120
pilot87	2031	4883	73804
stocfor3	16676	15695	74004
woodw	1099	8405	37478

ble 11.1, while the problem statistics of the 6 selected problems in Table 11.2. Finally, Table 11.3 shows what percentage of the solution time was spent in the different algorithmic units. The solution strategy in all cases was scaling, crash basis, dynamic partial pricing with 4 vectors in multiple pricing. Had some other strategy been used the results would look rather differently.

Table 11.3. Time percentages of algorithmic components of primal SSX using dynamic partial pricing with 4 vectors in multiple pricing.

Alg. unit	bnl2	fit2p	maros-r7	pilot87	stocfor3	woodw
CHKFEZ	3.1	0.1	0.2	0.3	2.5	1.6
BTRAN	16.9	17.8	29.0	14.4	21.1	15.1
PRICE	16.0	17.3	10.8	3.1	8.1	37.6
FTRAN	28.2	44.4	39.9	30.2	34.1	16.1
CHUZRO	18.6	14.6	6.9	3.5	19.1	12.1
FACTOR	2.5	1.1	6.5	40.3	2.5	1.9
HUZKEP	8.1	1.7	2.2	1.0	7.7	5.0
OTHERS	6.6	3.0	4.5	7.2	4.9	10.6

Apparently, the main time consuming units are BTRAN, PRICE, FTRAN, CHUZRO and FACTOR. Even they show large variations over the problems. The ratios of the smallest and largest figures are 1:2 (BTRAN), 1:12 (PRICE), 1:3 (FTRAN), 1:5 (CHUZRO) and 1:37 (FACTOR).

These variations clearly show why it is so difficult to create a scalable parallel version of the simplex method. Whatever the differences are among the relative times of the highlighted components it is clear that the above algorithmic units consume most of the solution time. Therefore, the overall efficiency of an implementation depends mainly on the quality of implementation of these units. This explains why so much effort has been spent on improving them.

Using the number of iterations to measure efficiency is not really meaningful. Experience shows that the least number of iterations is obtained if the incoming variable is selected on the basis of the largest achievable improvement in each iteration. However, this technique requires a ratio test with each profitable candidate which is nearly as tough as maintaining the full transformed tableau and is, therefore, prohibitive for any practical problem. On the other extreme, if we are satisfied with fast iterations (like choosing the first candidate) we may end up with a very large number of iterations and large solution time. The trade-off is somewhere in between. This is why so many pricing strategies have been designed.

Profiling is typically the tool of the developers of simplex implementations. As such it falls into the scope of this book. It is used for testing

the changes caused by algorithmic developments or different setting of parameters.

11.3. Some alternative techniques

In this section we briefly outline two interesting ideas that can be useful in some realistic cases.

11.3.1 Superbasic variables

Throughout the presentation of the variants of the simplex method it has been explicitly or implicitly assumed that the nonbasic variables are at one of their finite bounds and free nonbasic variables are kept at zero level. There was one exception from this practice, namely in the EXPAND procedure. Here, the outgoing variables could leave the basis at a small infeasible value. The purpose of this treatment was to enable better, in this case nondegenerate, iterations. This idea can further be generalized and nonbasic variables can be allowed to take any value different from their finite bounds. In practice it is usually a feasible value, i.e., $\ell_j < x_j < u_j$ for $j \in \mathcal{R}$. Such an x_j is called a *superbasic variable*. This notion was originally introduced by Murtagh in [Murtagh, 1981] for nonlinear programming but later it got adopted in LP, too.

Obviously, superbasic variables are set up only if some advantage can be derived from their introduction. One typical situation of this type can be when we want to reduce the degree of degeneracy of the basis. For instance, if the starting basis is too degenerate and also infeasible, one or more nonbasic variables with relatively dense columns can be assigned some feasible value. It entails the transformation of the RHS (see (1.13)) which can affect the degenerate positions. As a side effect the infeasibility of the basis can deteriorate. However, it is now viewed as less of a problem than degeneracy.

Superbasic variables can be priced as free variables during simplex iterations, namely, they can go up or down. Therefore, if their reduced cost is different from zero they are entering candidates. Obviously, when determining the steplength their true bounds must be observed. If the superbasic variables were set up to reduce degeneracy then it is not advisable to enter them into the basis too soon as they may restore the degenerate situation.

When the simplex method stops and there are still some superbasic variables left (with zero reduced costs, of course) they can be moved to a finite bound to obtain a basic solution in the mathematical sense. This move, however, may make the basis infeasible which must be elim-

inated at the expense of corrective iterations (probably using the dual algorithm).

11.3.2 Row basis

It is well known that the computational effort of a simplex iteration with dense entities (matrix and vectors) is roughly proportional to m^2 . If sparse computing is used the number of nonzeros plays the defining role in the numerical computations but the size of the basis (m) still determines the amount of administrative work and some auxiliary arrays which may not be negligible. There is an algorithmic possibility to use the simplex method with a working basis of size $\min\{m, n\}$ where n now denotes the number of structural variables. If n defines the minimum we can use the *row basis* technique which we outline below.

The following discussion is not a rigorous development, though it contains all the important elements of the row basis technique. To demonstrate the idea it is sufficient to consider the LP problem in the following form

$$(P2) \quad \text{minimize} \quad z = \mathbf{c}^T \mathbf{x} \quad (11.1)$$

$$\text{subject to} \quad \mathbf{A}\mathbf{x} \geq \mathbf{b} \quad (11.2)$$

$$\mathbf{x} \geq \mathbf{0}, \quad (11.3)$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$, the vectors are of compatible dimensions and $m > n$, i.e., there are more constraints than structural variables. Assume \mathbf{A} is of full column rank. Let \mathbf{x} be a feasible solution to (P2) such that it satisfies n linearly independent constraints as equalities. It can be verified that this \mathbf{x} is a vertex of feasible region of (P2). The corresponding constraints are said to be *active* while the remaining ones are *inactive*. It can be assumed that the active constraints are permuted to the first n rows. The matrix of the tight constraints is nonsingular (having linearly independent rows) which can be viewed as a basis $\mathbf{B} \in \mathbb{R}^{n \times n}$. In this way, \mathbf{A} and \mathbf{b} can be partitioned as

$$\mathbf{A} = \begin{bmatrix} \mathbf{B} \\ \mathbf{R} \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_{\mathcal{B}} \\ \mathbf{b}_{\mathcal{R}} \end{bmatrix},$$

where $\mathcal{B} = \{1, \dots, n\}$ and $\mathcal{R} = \{n+1, \dots, m\}$ are used also as subscripts to identify subvectors. \mathbf{B} is now considered in a rowwise form

$$\mathbf{B} = \begin{bmatrix} \mathbf{a}^1 \\ \vdots \\ \mathbf{a}^n \end{bmatrix}$$

With the given \mathbf{x} the constraints in (11.2) can be written as follows:

$$\mathbf{Bx} = \mathbf{b}_B, \quad \text{from which } \mathbf{x} = \mathbf{B}^{-1}\mathbf{b}_B$$

and

$$\mathbf{Rx} \geq \mathbf{b}_R.$$

We call this \mathbf{x} a basic feasible solution. If $\mathbf{Rx} > \mathbf{b}_R$ holds we say that \mathbf{x} is nondegenerate, otherwise, if at least one constraint in \mathbf{R} holds as equality \mathbf{x} is degenerate.

For the rest of the discussion it is assumed that \mathbf{x} is a nondegenerate basic feasible solution. To be able to determine the optimality conditions in terms of the above representation some notations have to be introduced. Let \mathbf{y} be defined to satisfy

$$\mathbf{y}^T \mathbf{B} = \mathbf{c}^T, \quad (11.4)$$

from where $\mathbf{y}^T = \mathbf{c}^T \mathbf{B}^{-1}$. It is easy to see that (11.4) can also be written as

$$\mathbf{c} = \mathbf{B}^T \mathbf{y} = \sum_{i=1}^n y_i \mathbf{a}^i, \quad (11.5)$$

where \mathbf{a}^i is row i of \mathbf{A} (and also \mathbf{B}).

To see whether the objective value corresponding to \mathbf{x} can be improved we investigate the effects of feasible movements from \mathbf{x} . To do so, the p -th equality constraint $\mathbf{a}^p \mathbf{x} = b_p$ is relaxed to $\mathbf{a}^p \mathbf{x} > b_p$ such that all other active constraints remain tight. The direction of movement is denoted by \mathbf{w} and the level of relaxation of constraint p by t . We want \mathbf{w} to satisfy

$$\mathbf{a}^i(\mathbf{x} + t\mathbf{w}) = b_i, \quad i \in \mathcal{B} \setminus \{p\}, \quad \text{and} \quad \mathbf{a}^p(\mathbf{x} + t\mathbf{w}) > b_p$$

with $t < 0$. These requirements imply that for this \mathbf{w}

$$\mathbf{a}^i \mathbf{w} = 0, \quad i \in \mathcal{B} \setminus \{p\}, \quad (11.6)$$

$$\mathbf{a}^p \mathbf{w} > 0. \quad (11.7)$$

must hold to maintain feasibility. We assume $|t|$ is small enough so that all inactive constraints remain satisfied at $\mathbf{x} + t\mathbf{w}$. Quick calculation gives that the objective value in $\mathbf{x} + t\mathbf{w}$ is $\mathbf{c}^T \mathbf{x} + t y_p \mathbf{a}^p \mathbf{w}$. Thus the change in the objective function is

$$\Delta z = \mathbf{c}^T(\mathbf{x} + t\mathbf{w}) - \mathbf{c}^T \mathbf{w} = t y_p \mathbf{a}^p \mathbf{w}. \quad (11.8)$$

From (11.7) we know that $\mathbf{a}^p \mathbf{w} > 0$. As $t < 0$, and the objective is to be minimized, $\Delta z < 0$ can be achieved if $y_p > 0$. Therefore, we can

say that \mathbf{x} is an optimal basic feasible solution if there is no improving direction, i.e., if

$$\mathbf{y} \leq \mathbf{0} \quad (11.9)$$

for \mathbf{y} defined in (11.4). (11.9) is the optimality condition for the row basis representation.

If (11.9) is not satisfied there is at least one $i \in \mathcal{B}$ such that $y_i > 0$. In this case we can determine a direction of improvement by noticing that the actual size of \mathbf{w} is not important but only the sign of $\mathbf{a}^p \mathbf{w}$. Therefore, if we replace (11.7) by $\mathbf{a}^p \mathbf{w} = \text{sign}(y_s)$, (11.6) and (11.7) can be replaced by

$$\mathbf{Bw} = \text{sign}(y_s) \mathbf{e}_p \quad (11.10)$$

which then has to be solved for \mathbf{w} to obtain an improving direction.

To make the largest progress in the objective function, $|t|$ has to be made as large as possible. As the basic equations remain satisfied because of (11.6) the magnitude of t is restricted only by the feasibility of the inactive constraints. It says $\mathbf{a}^j(\mathbf{x} + t\mathbf{w}) \geq b_j$ must remain satisfied for $j \in \mathcal{R}$. From this,

$$t \mathbf{a}^j \mathbf{w} \geq b_j - \mathbf{a}^j \mathbf{x}. \quad (11.11)$$

The right-hand side of (11.11) is negative because it is derived from a satisfied nondegenerate constraint in \mathcal{R} . If $\mathbf{a}^j \mathbf{w} \leq 0$ then relation (11.11) always holds. If $\mathbf{a}^j \mathbf{w} > 0$ then t is limited by

$$t \geq \frac{b_j - \mathbf{a}^j \mathbf{x}}{\mathbf{a}^j \mathbf{w}}, \quad \text{if } \mathbf{a}^j \mathbf{w} > 0. \quad (11.12)$$

All constraints in \mathcal{R} remain satisfied if t is chosen to be

$$\theta_R = \max_{j \in \mathcal{R}} \left\{ \frac{b_j - \mathbf{a}^j \mathbf{x}}{\mathbf{a}^j \mathbf{w}}, \mathbf{a}^j \mathbf{w} > 0 \right\}. \quad (11.13)$$

If $\mathbf{a}^j \mathbf{w} \leq 0$ for all $j \in \mathcal{R}$ the problem is unbounded. Otherwise, let the index of the constraint that defines θ_R be denoted by q . At $\mathbf{x} + \theta_R \mathbf{w}$ constraint q becomes tight and can replace the relaxed constraint p in the basis. With this change the new \mathbf{B} remains nonsingular (because of $\mathbf{a}^j \mathbf{w} \neq 0$). θ_R is the steplength with the row basis.

The above outlined steps can easily be refined to make a legitimate algorithm out of them. It is an alternative version of the simplex method. Its generalization for problems with all types of variables and constraints is also a straightforward task for the readers of this book.

If we have a problem with $m \gg n$ the advantages of working with a smaller basis can be substantial. Therefore, in a demanding implementation it is worth including the row basis version of the simplex method in

both primal and dual form (the latter has not been discussed here). Another (not necessarily easy) possibility is to form the dual of the problem after input and preprocessing and solve it with the primal simplex.

Readers interested in some further details of the row basis technique are referred to Nazareth's book [Nazareth, 1987].

11.4. Modeling systems

Real-world LP and MILP problems tend to be very large. They have to be formulated, edited, verified and maintained. All these functions call for a *model management system*. Modeling languages have been created to make the above activities possible in a convenient and efficient way.

A key requirement is to allow the formulation of a model in a similar way as humans naturally do it. This enables better understanding of the model and also serves as a documentation of the work done.

Several modeling languages have been developed and implemented. They all share some common features. Here are some of the most important ones.

- Model building is supported by a *model editor*.
- Separation of model and data.
- Variables and constraints are identified by meaningful names.
- Sophisticated indexing of variables is possible.
- Variables can be placed on both sides of the constraints.
- Algebraic expressions can be used for defining constraints.
- Logical relationships can be defined.
- Data can be imported.
- Macros can be defined for repetitive parts of the model.
- Sub-models can be integrated into a large model.
- Comments can be inserted anywhere in the model description.

The output of the model editor is a file that forms the input of an LP solver. Several formats for the output can be chosen. The most important one is the widely used industry standard MPS input format for linear programming. It is to be noted that the MPS format has been extended to be used for (mixed) integer and, recently, by Maros and Mészáros [Maros and Mészáros, 1999], for quadratic programming problems.

Modeling systems often serve as a model development environment. They are able to invoke solvers directly and can also retrieve the solution for further processing.

The different modeling systems are commercial products and their survey is beyond the scope of this book.

11.5. A prototype simplex solver

The material covered so far is sufficient for a high quality implementation of the simplex method. The discussed algorithmic units can be used as building blocks. To create a system out of them the ideas of design and implementation of optimization software (Chapter 4) should be observed. In particular, modularity is important for the gradual development and extension.

The design of an implementation always starts with some decisions on what algorithmic features to include. It is followed by designing the appropriate data structures.

In this section we give the structure of a simplex based prototype LP system. An advanced implementation must contain both the primal and the dual methods. Also, smooth transition from primal to dual iterations (and vice versa) should be made possible. We have pointed out in several cases that the combination of the two main algorithms (one followed by the other to finish off or to make a correction) can result in the best performance. If the system is to be used for solving MILP problems both algorithms are needed, anyhow. It is assumed that the run parameters, including the name of the problem file are specified in a parameter file. The following functional units must be present.

Fast MPS input. Converts a problem into internal representation (see Chapter 6). If memory allows it prepares a row- and columnwise storage of \mathbf{A} . Provides statistics about the problem (m , \bar{n} , ν_A , number of variables and constraints by type, density of \mathbf{A} , number of nonzeros in the objective row, right-hand side and range).

Advanced versions of this functional unit are able to analyze the problem and suggest a solution strategy. They can also display the nonzero pattern of \mathbf{A} with zoom facility and produce further statistics about the distribution of magnitudes of the nonzero elements. Visual inspection of the matrix can help select a good pricing strategy.

Preprocessing. Presolve and scaling should be made optional. The level of presolve (the actual tests to be performed) can also be made optional. Different versions of scaling (section 7.2) can be included to provide a choice. The IBM method of Benichou et. al., is a good

candidate if the problem is stable enough to tolerate the small round-off errors generated (the majority of the problems are such). However, for numerically demanding problems the Curtis-Reid scaling is better and should also be made available. Finally, it is important to allow the option of ‘no scaling’ as most combinatorial problems are best solved in this way.

Starting procedures. The absolute minimum is the lower logical basis. Additionally, at least one of the crash techniques (section 9.8.2), generally CRASH(LTSF) or a variant of it must also be available. It makes sense to implement an upper triangular crash based on the same ideas as LTSF, which could be called UTSF. In this case there is no lower triangular part and the LU form update can be very efficient, at least at the beginning of the iterations.

Factorization (LU), section 8.2, or inversion (PFI), section 8.1. The LU form is highly preferred because of its proven superiority. As the PFI is easier to implement and performs quite well it can be the choice for a first working version of an LP system. Great care must be exercised when designing the data structures to enable the replacement of PFI by LU at a later stage.

Updating triangular factors for the LU version, section 8.2.3. While the product form update is also a realistic alternative for the LU the triangular update is more efficient. It leads to a slower growth of nonzeros which, in turn, reduces the need for too frequent refactorizations. If the product form update is used then the basis has to be refactorized after every 30–50 iterations. For the triangular update it goes up to 100–150 iterations.

Pricing strategies. They are slightly different for the primal and the dual. In primal the important design decision is whether or not to maintain an explicit row vector of reduced costs. If memory allows it is recommended to keep \mathbf{d} and update it after every basis change. This applies to primal phase-2. As the composition of the objective function keeps changing in primal phase-1 it is reasonable to recompute the corresponding reduced costs as updating would be too complicated. For dual phase-2 the dual reduced costs (the primal basic variables) are explicitly available. The dual phase-1 reduced costs are usually recomputed as the structure of the dual objective function can change in every iteration.

Regarding the selection of the incoming (in primal) or outgoing (in dual) variables, the recommendation is to implement (nearly) all of

the discussed methods (sections 9.5 and 10.4). For a gradual build-up in the primal the order can be multiple pricing with up to 6 vectors, sectional pricing (it can be an absolute winner if the parameters are properly chosen, see visualization of \mathbf{A}), then some normalized pricing (at least Devex or ASE but possibly also steepest edge). For dual, after the obvious (and not very efficient) Dantzig pricing, Devex and steepest edge should be included. A good idea is to implement the general pricing framework SIMPRI of section 9.5.5 and control it by setting its parameters to obtain the required pricing method.

The availability of a range of pricing algorithms makes it possible to choose one that fits the actual problem the best. In an ideal world this choice can be made by a supervisor program, see ‘Control unit’ later.

Good default strategies for the primal are either dynamic partial pricing (see section 9.5.2) combined with multiple pricing with four vectors ($H = 4$) or Devex and Devex for the dual.

Column and row update. The selected column(s) (in primal) or rows (in dual) have to be updated. This is a computationally demanding procedure. It is important to be able to exploit sparsity of the vectors involved. We have pointed out that the row- and columnwise storage of \mathbf{A} and a bit more sophisticated double storage of \mathbf{U} of the LU decomposition are the main sources of speed in this part of the simplex method (section 8.2).

If either the simplex multiplier $\boldsymbol{\pi}$ or the vector of reduced costs \mathbf{d} is kept updated it should be transformed as described in sections 9.4.2 and 9.4.3, respectively.

Pivot methods. This is actually the ratio test according to the main algorithm (primal or dual) and the feasibility (phase-1 or -2) for determining the outgoing (in primal) or incoming (in dual) variable. All these methods must be implemented carefully as they affect the numerical stability and degeneracy of the solution.

Control unit. Its first task is to interpret the parameter file, if present. It monitors and supervises the entire solution process. Invokes functional units, evaluates their outcome. Makes decision on the algorithmic techniques. Triggers refactorization (reinversion). Changes solution strategy if necessary (e.g., checks whether an anti-degeneracy strategy has to be applied, makes decision on pricing strategy). It can be quite sophisticated in case of an implementation that has a considerable algorithmic richness.

Solution output. The solution must be presented in the ‘unresolved’ form which is usually obtained after the appropriate postsolve procedure has been applied. Complete primal and dual solutions are to be sent to a file for possible further processing. Usually one line for each variable and constraint is prepared containing some replicated input data (name, variable/row type, lower and upper bound, objective coefficient/RHS value, range) and computed values (primal and dual variables). It is useful to include some additional information by marking basic variables and variables at bound. If the solution is infeasible variables at infeasible level (in \mathcal{P} or \mathcal{M}) could also be indicated. If a modeling system was used this file is probably retrieved by it. Otherwise, the file can be perused or submitted to a special purpose program for further analysis, processing and presentation.

It is useful if a sensitivity analysis can be performed on the solution obtained. Though this is book does not cover the topic it is worth knowing what it can do. Broadly speaking it investigates the relationship between the current optimal solution and some small changes in the problem data to determine how much a chosen data item can change so that the basis remains optimal. It can help make a statement about the robustness of the solution.

A facility should be provided to monitor the progress of optimization at different levels. Possibilities are: (i) no output until termination, (ii) one line after every specified number of iterations, or refactorization (reinversion), (iii) one line per major iteration, (iv) one line per minor iteration or even a follow through of the actions within each iteration. The log could optionally go to the screen and/or a file.

References

- [Adler et al., 1989] Adler, I., Karmarkar, N., Resende, M., and Veiga, G. (1989). Data Structures and Programming Techniques for the Implementation of Karmarkar's Algorithm. *ORSA Journal on Computing*, 1:84–106.
- [Andersen and Andersen, 1995] Andersen, E. and Andersen, K. (1995). Presolving in Linear Programming. *Mathematical Programming*, 71(2):221–245.
- [Bartels and Golub, 1969] Bartels, R. and Golub, G. (1969). The Simplex Method of Linear Programming Using LU Decomposition. *Communications of the ACM*, pages 266–268.
- [Beale, 1968] Beale, E. (1968). *Mathematical Programming in Practice*. Topics in operational research. Pitman & Sons Ltd, London.
- [Benichou et al., 1977] Benichou, M., Gautier, J., Hentges, G., and Ribiere, G. (1977). The efficient solution of large-scale linear programming problems. *Mathematical Programming*, 13:280–322.
- [Bisschop and Meeraus, 1977] Bisschop, J. and Meeraus, A. (1977). Matrix Augmentation and Partitioning in the Updating of the Basis Inverse. *Mathematical Programming*, 13:241–254.
- [Bixby, 1992] Bixby, R. (1992). Implementing the Simplex Method: The Initial Basis. *ORSA Journal on Computing*, 4(3):267–284.
- [Bland, 1977] Bland, R. (1977). New finite pivot rule for the simplex method. *Mathematics of Operations Research*, 2:103–107.
- [Brearley et al., 1975] Brearley, A., Mitra, G., and Williams, H. (1975). Analysis of Mathematical Programming Problems Prior to Applying the Simplex Method. *Mathematical Programming*, 8:54–83.
- [Brown and Olson, 1993] Brown, G. and Olson, M. (1993). Dynamic factorization in large scale optimization. Technical Report NPSOR-93-008, Naval Postgraduate School, Monterey, California.
- [Charnes, 1952] Charnes, A. (1952). Optimality and degeneracy in linear programming. *Econometrica*, 20:160–170.

- [Chinneck, 1997] Chinneck, J. (1997). Computer codes for the analysis of infeasible linear programs. *Journal of Operations Research Society*, 47(1):61–72.
- [Chvátal, 1983] Chvátal, V. (1983). *Linear Programming*. Freeman Press, New York.
- [Cunningham, 1979] Cunningham, W. (1979). Theoretical Properties of the Network Simplex Method. *Mathematics of Operations Research*, 4(2):196–208.
- [Curtis and Reid, 1972] Curtis, A. and Reid, J. (1972). On the automatic scaling of matrices for Gaussian elimination. *Journal of the Institute of Mathematics and its Applications*, 10:118–124.
- [Dantzig, 1963] Dantzig, G. (1963). *Linear Programming and Extensions*. Princeton University Press, Princeton.
- [Dantzig, 1988] Dantzig, G. (1988). Impact of linear programming on computer development. *OR/MS Today*, pages 12–17.
- [Dantzig and Orchard-Hays, 1953] Dantzig, G. and Orchard-Hays, W. (1953). Notes on linear programming: Part V. — Alternative algorithm for the revised simplex method using product form of the inverse. Research Memorandum RM-1268, The RAND Corporation.
- [Dantzig and Orchard-Hays, 1954] Dantzig, G. and Orchard-Hays, W. (1954). The product form of the inverse in the simplex method. *Mathematical Tables and Other Aids to Computation*, 8:64–67.
- [Dantzig et al., 1955] Dantzig, G., Orden, A., and Wolfe, P. (1955). A generalized simplex method for minimizing a linear form under linear inequality constraints. *Pacific Journal of Mathematics*, 5(2):183–195.
- [Dobosy, 1970] Dobosy, A. (1970). Fotri algorithm. Private communication.
- [Duff et al., 1986] Duff, I., Erisman, A., and Reid, J. (1986). *Direct Methods for Sparse Matrices*. Oxford Science Publications. Clarendon Press, Oxford.
- [Erisman et al., 1985] Erisman, A., Grimes, R., Lewis, J., and Poole, Jr., W. (1985). A structurally stable modification of Hellerman-Rarick's P^4 algorithm for reordering unsymmetric sparse matrices. *SIAM Journal on Numerical Analysis*, 22:369–385.
- [Forrest and Goldfarb, 1992] Forrest, J. and Goldfarb, D. (1992). Steepest edge simplex algorithms for linear programming. *Mathematical Programming*, 57(3):341–374.
- [Forrest and Tomlin, 1972] Forrest, J. and Tomlin, J. (1972). Updating triangular factors of the basis to maintain sparsity in the product-form simplex method. *Mathematical Programming*, 2:263–278.
- [Fourer, 1994] Fourer, R. (1994). Notes on the dual simplex method. Unpublished.
- [Fourer and Gay, 1993] Fourer, R. and Gay, D. (1993). Experience with a Primal Pre-solve Algorithm. Numerical Analysis Manuscript 93–06, AT&T Bell Laboratories, Murray Hill, NJ, USA.

- [Fulkerson and Wolfe, 1962] Fulkerson, D. and Wolfe, P. (1962). An algorithm for scaling matrices. *SIAM Review*, 4:142–146.
- [Gal, 1990] Gal, T. (1990). Degeneracy in mathematical programming and degeneracy graphs. *ORiON*, 6:3–36.
- [Gay, 1985] Gay, D. (1985). Electronic mail distribution of linear programming test problems. *COAL Newsletter*, 13:10–12.
- [George and Liu, 1981] George, A. and Liu, W.-H. (1981). *Computing Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, N.J.
- [Gill et al., 1989] Gill, P., Murray, W., Saunders, M., and Wright, M. (1989). A Practical Anti-Cycling Procedure for Linearly Constrained Optimization. *Mathematical Programming*, 45:437–474.
- [Goldfarb, 1977] Goldfarb, D. (1977). On the Bartels-Golub decomposition for linear programming bases. *Mathematical Programming*, 13:272–279.
- [Goldfarb and Reid, 1977] Goldfarb, D. and Reid, J. (1977). A Practicable Steepest-Edge Simplex Algorithm. *Mathematical Programming*, 12:361–371.
- [Gondzio, 1997] Gondzio, J. (1997). Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method. *INFORMS Journal on Computing*, 9(1):73–91.
- [Gould and Reid, 1989] Gould, N. and Reid, J. (1989). New crash procedures for large systems of linear constraints. *Mathematical Programming*, 45:475–501.
- [Greenberg, 1978a] Greenberg, H., editor (1978a). *Design and Implementation of Optimization Software*, NATO ASI. Sijthoff and Nordhoff.
- [Greenberg, 1978b] Greenberg, H. (1978b). Pivot selection tactics. In [Greenberg, 1978a], pages 143–174.
- [Greenberg, 1978c] Greenberg, H. (1978c). A tutorial on matricial packing. In [Greenberg, 1978a], pages 109–142.
- [Greenberg, 1983] Greenberg, H. (1983). A functional description of ANALYZE: A computer assisted analysis system for linear programming models. *ACM Transactions on Mathematical Software*, 9:18–56.
- [Greenberg and Kalan, 1975] Greenberg, H. and Kalan, J. (1975). An exact update for Harris' tread. *Mathematical Programming Study*, 4:26–29.
- [Gustavson, 1972] Gustavson, F. (1972). Some basic techniques for solving sparse systems of linear equations. In [Rose and Willoughby, 1972], pages 41–52.
- [Hadley, 1962] Hadley, G. (1962). *Linear Programming*. Addison-Wesley.
- [Hamming, 1971] Hamming, R. (1971). *Introduction to applied numerical analysis*. McGraw-Hill, New York.
- [Harris, 1973] Harris, P. (1973). Pivot Selection Method of the Devex LP Code. *Mathematical Programming*, 5:1–28.

- [Hellerman and Rarick, 1971] Hellerman, E. and Rarick, D. (1971). Reinversion with the preassigned pivot procedure. *Mathematical Programming*, 1:195–216.
- [Hellerman and Rarick, 1972] Hellerman, E. and Rarick, D. (1972). The partitioned preassigned pivot procedure (P4). In [Rose and Willoughby, 1972], pages 67–76.
- [Hillier and Lieberman, 2000] Hillier, F. and Lieberman, G. (2000). *Introduction to Operations Research*. McGraw-Hill, 7th edition.
- [Kalan, 1971] Kalan, J. (1971). Aspects of Large-Scale In-Core Linear Programming. In *Proceedings of the 1971 annual conference of the ACM*, pages 304–313. ACM.
- [Khaliq, 1997] Khaliq, M. (1997). An Efficient Input Algorithm For Large Scale Optimization. MEng final year project, Department of Computing, Imperial College, London.
- [Klee and Minty, 1972] Klee, V. and Minty, G. (1972). How good is the simplex algorithm? In Shisha, O., editor, *Inequalities III*, pages 159–175. Academic Press, New York.
- [Knuth, 1968] Knuth, D. (1968). *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts.
- [Lemke, 1954] Lemke, C. (1954). The Dual Method of Solving the Linear Programming Problem. *Naval Research Logistics Quarterly*, 1:36–47.
- [Liesegang, 1980] Liesegang, G. (1980). Aggregation in linear optimization models. Habilitation thesis, Cologne. in German.
- [Markowitz, 1957] Markowitz, H. (1957). The Elimination Form of the Inverse and its Applications to Linear Programming. *Management Science*, 3:255–269.
- [Maros, 1981] Maros, I. (1981). Adaptivity in linear programming, II. *Alkalmazott Matematikai Lapok (Journal of Applied Mathematics)*, 7:1–71. In Hungarian.
- [Maros, 1986] Maros, I. (1986). A general Phase-I method in linear programming. *European Journal of Operational Research*, 23:64–77.
- [Maros, 1990] Maros, I. (1990). On pricing in network LP. In *ARIDAM V Abstracts*, number 2–90 in RUTCOR Reports, Rutgers University, NJ, USA. RUTCOR.
- [Maros, 2002a] Maros, I. (2002a). A General Pricing Scheme for the Simplex Method. *Annals of Operations Research*. To appear.
- [Maros, 2002b] Maros, I. (2002b). A Generalized Dual Phase-2 Simplex Algorithm. *European Journal of Operational Research*. To appear.
- [Maros, 2002c] Maros, I. (2002c). A Piecewise Linear Dual Phase-1 Algorithm for the Simplex Method With All Types of Variables. *Computational Optimization and Applications*. To appear.
- [Maros and Khaliq, 2002] Maros, I. and Khaliq, M. (2002). Advances in Design and Implementation of Optimization Software. *European Journal of Operational Research*, 140(2):322–337.

- [Maros and Mészáros, 1999] Maros, I. and Mészáros, C. (1999). A repository of convex quadratic programming problems. *Optimization Methods and Software*, 11&12:671–681.
- [Maros and Mitra, 2000] Maros, I. and Mitra, G. (2000). Investigating the Sparse Simplex Algorithm on a Distributed Memory Multiprocessor. *Parallel Computing*, 26:151–170.
- [Murtagh, 1981] Murtagh, B. (1981). *Advanced Linear Programming: Computation and Practice*. McGraw-Hill.
- [Nazareth, 1987] Nazareth, J. (1987). *Computer Solution of Linear Programs*. Oxford University Press, New York, Oxford.
- [Orchard-Hays, 1968] Orchard-Hays, W. (1968). *Advanced Linear-Programming Computing Techniques*. McGraw-Hill, New York.
- [Orchard-Hays, 1978] Orchard-Hays, W. (1978). History of mathematical programming systems. In [Greenberg, 1978a], pages 1–26.
- [Padberg, 1995] Padberg, M. (1995). *Linear Optimization and Extensions*. Springer.
- [Prékopa, 1968] Prékopa, A. (1968). *Linear Programming*. János Bolyai Mathematical Society, Budapest. in Hungarian.
- [Prékopa, 1995] Prékopa, A. (1995). *Stochastic Programming*. Kluwer Academic Publishers and Akadémiai Kiadó.
- [Prékopa, 1996] Prékopa, A. (1996). A brief introduction to linear programming. *Math. Scientist*, 21:85–111.
- [Reid, 1971] Reid, J. (1971). A Note on the Stability of Gaussian Elimination. *Journal of the Institute of Mathematics and Its Application*, 8:374–375.
- [Reid, 1982] Reid, J. (1982). A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases. *Mathematical Programming*, 24(1):55–69.
- [Roos et al., 1997] Roos, C., Terlaky, T., and Vial, J.-P. (1997). *Theory and Algorithms for Linear Optimization*. Discrete Mathematics and Optimization. Wiley.
- [Rose and Willoughby, 1972] Rose, D. and Willoughby, R., editors (1972). *Sparse Matrices and Their Applications*. Plenum Press.
- [Ryan and Osborne, 1988] Ryan, D. and Osborne, M. (1988). On the Solution of Highly Degenerate Linear Programmes. *Mathematical Programming*, 41:385–392.
- [Saunders, 1976] Saunders, M. (1976). A fast and stable implemetation of the simplex method using Bartels-Golub updating. In Bunch, J. and Rose, D., editors, *Sparse Matrix Computation*, pages 213–226. Academic Press, New York.
- [Sedgewick, 1990] Sedgewick, R. (1990). *Algorithms in C*. Addison-Wesley.

- [Suhl and Suhl, 1990] Suhl, U. and Suhl, L. (1990). Computing Sparse LU Factorizations for Large-Scale Linear Programming Bases. *ORSA Journal on Computing*, 2(4):325–335.
- [Suhl and Suhl, 1993] Suhl, U. and Suhl, L. (1993). A fast LU update for linear programming. *Annals of Operations Research*, 43:33–47.
- [Swietanowski, 1998] Swietanowski, A. (1998). A New Steepest Edge Approximation for the Simplex Method for Linear Programming. *Computational Optimization and Applications*, 10(3):271–281.
- [Terlaky, 1996] Terlaky, T., editor (1996). *Interior Point Methods of Mathematical Programming*, volume 5 of *Applied Optimization*. Kluwer Academic Publishers.
- [Tomlin, 1974] Tomlin, J. (1974). On pricing and backword transformation in linear programming. *Mathematical Programming*, 6:42–47.
- [Tomlin, 1975] Tomlin, J. (1975). On scaling linear programming problems. *Mathematical Programming Study*, 4:146–166.
- [Tomlin and Welch, 1983a] Tomlin, J. and Welch, J. (1983a). Formal oprimization of some reduced linear programming problems. *Mathematical Programming*, 27(2):232–240.
- [Tomlin and Welch, 1983b] Tomlin, J. and Welch, J. (1983b). A pathological case in the reduction of linear programmings. *Operations Research Letters*, 2:53–57.
- [Tomlin and Welch, 1986] Tomlin, J. and Welch, J. (1986). Finding duplicate rows in a linear program. *Operations Research Letters*, 5:7–11.
- [Vanderbei, 2001] Vanderbei, R. J. (2001). *Linear Programming: Foundations and Extensions*, volume 37 of *International Series in Operations Research and Management Science*. Kluwer Academic Publishers, Boston, 2nd edition.
- [Wilkinson, 1965] Wilkinson, J. (1965). *The Algebraic Eigenvalue Problem*. Monographs on Numerical Analysis. Clarendon Press, Oxford.
- [Wolfe, 1963] Wolfe, P. (1963). A technique for resolving degeneracy in linear programming. *SIAM Journal of Applied Mathematics*, 11:205–211.
- [Wolfe, 1965] Wolfe, P. (1965). The composite simplex algorithm. *SIAM Review*, 7(1):42–54.

Index

- Active constraint, 307–308
- Active submatrix, 125, 138
- ADACOMP, 228–230
- Adler, 85
- Aggregation, 98
- Alternative optima, 20, 162
- ALU, 63
- Andersen, 98
- Arithmetic logic unit, 63
- Arithmetic mean, 113
- Artificial Elimination, 36
- ASE, 197
- Bartels, 149
- Basis heading, 129, 162
- Basis, 21
 - CPLEX, 253, 255
 - crash, 246
 - dual feasible, 294
 - initial, 244, 259
 - logical, 245–246
 - lower, 245, 312
 - mixed, 245
 - upper, 245
 - neighboring, 26–27, 41
 - partial, 260
 - starting, 244, 246, 306
 - triangular, 246
 - lower, 246
 - upper, 246
 - trivial, 245
- Benichou, 196, 242
- Big-M method, 45
- Binary search, 95
- Bixby, 253
- Bland's rule, 203, 230
- Bound flip, 172, 177, 181, 212, 268–271, 274
- Bound translation, 10, 17, 101
- Branch prediction, 64
- Branching, 64
- Break point, 207–208, 211, 213, 222, 273, 275, 281, 284, 287
 - multiple, 270
 - multiplicity, 208, 273
- Brearley, 98
- BTRAN, 132–133, 149, 155–156, 182–183, 192, 195–196, 233, 237, 304–305
- BTRANL, 149, 156–157
- BTRANU, 149, 156, 158
- C/C++, 65–66
- Cache hit, 62
- Cache miss, 63
- Cache, 62, 70
- Capacity, 60
- Charnes, 230
- Chinneck, 37
- Choose Column, 33
- CHUZRO, 33, 177, 304–305
- Chvatal, 265
- CISC, 63
- Clark, 47
- Code stability, 61
- Computational form #1, 13, 20, 40, 161
- Computational form #2, 17, 161
- Computer science, xviii
- Computer technology, xviii
- Constraint type, 6
- Constraints, 3
- Convex set, 23
- Cost coefficient, 3
- CPU, 62–63
- Crash procedure, 248
 - anti-degeneracy, 251
 - CRASH(ADG), 251–253
 - CRASH(LTS), 248
 - CRASH(LTSF), 248, 250–254, 312
 - numerical, 248
 - symbolic, 248
 - lower triangular, 248
- Crusoe chip, 63

- Cunningham's rule, 203
- Curtis, 111–113
- Cycling, 32, 43, 230
- Dantzig, 3, 19, 187, 230
- Data structures, 69
 - dynamic, 69–70, 76
 - static, 69
- Decomposition algorithms, 52
- Default value, 301
- Degeneracy, 22, 164, 178, 224–225, 230, 235–237, 242–243, 245, 313
 - coping with, 178, 186, 232, 234
 - degree of, 22, 36, 164, 306
 - depth of, 235
 - dual, 43, 272–273, 276, 280, 287, 298
 - primal, 22, 24, 31
- Density, 50
- Design principles, 59
- Displacement, 30, 165–168, 170–173, 176, 206, 209, 214, 221, 224–225, 262
- Dot product, 74
- DSM-1, 44–45
- Dual feasibility correction, 278–279
- Dual-G, 266–269
- Dual-GX, 270–271, 273–274, 276, 291
- Dual-U, 265–267, 275, 286
- Duality, 38
- Duff, 85
- Dynamic branching, 64
- Ease of use, 61
- Efficiency, 60
- EFI, 55, 134
- Elementary transformation matrix, 27, 53, 123
- Elimination form of the inverse, 55, 134
- Elimination, 138
- Empty column, 100
- Empty row, 100
- Equality constraint, 7
- Equilibration, 112–113
- Erisman, 85, 258
- Error, 55, 178
 - accumulation, 55
 - cancellation, 55, 109
 - computational, 128
 - propagation, 55
 - rounding, 55, 109
- Eta, 27, 131, 157
- ETM, 27, 123–124, 131–133, 156–157
- EXPAND, 181, 231, 237–239, 243, 298
- Extensibility, 61
- External representation, 87
- Extreme direction, 24
- Extreme point, 23–25
- FACTOR, 305
- Factorization, 122, 312
- Feasibility range, 11
- Feasibility, 20, 40, 178
 - dual, 40, 261, 263
 - primal, 20
- FEWPHI, 214, 226
- Fill-in, 53, 55, 70–73, 127–129, 139–140, 152
- Floating point unit, 63
- Forcing constraint, 102
- Forrest, 149, 293
- Forrest-Tomlin update, 152
- FORTRAN, 65–66
- Fourer, 98, 109, 266
- FPU, 63
- FTRAN, 132–133, 149, 155–156, 198, 272, 295, 304–305
- FTRANL, 149, 156–157
- FTRANU, 149, 156–157
- Fulkerson, 111
- Full pivoting, 140
- Gal, 230
- Garbage collection, 144
- Gather, 71
- Gay, 51, 98, 109
- GDPO, 285–288, 290–291
- General constraints, 4, 14
- General form of LP, 4
- Geometric mean, 113
- Geometry of constraints, vii, 23
- George, 85
- Gill, 181, 237
- Global maximum, 209, 284
- Goldfarb, 149, 190, 293
- Golub, 149
- Gondzio, 98, 102–103, 107, 118
- Gould, 255
- Greek epsilon, 168
- Greenberg, 37, 85, 189, 195, 232
- Grimes, 258
- Gustavson, 85
- Hamming, 111
- Hard disk, 62
- Hardware, 62
- Harris, 179, 193, 296
- Hashing, 95
- Hessenberg matrix, 151
- Hillier, 47
- Hybrid processors, 63
- IEEE 754 standard, 63
- Implementation technology, 59
- Implementation, 59, 250, 311
- Inactive constraint, 307–309
- Individual bounds, 6
- Individual constraints, 4
- Interior point methods, xvii, 19
- Internal representation, 87, 94, 311
- Inverse, 27
- Inversion, 122, 312
- IPM, 19

- Joint constraints, 4
 Kalan, 50, 85, 189, 195
 Karmarkar, 85
 Kernel, 128, 136
 Khaliq, 60, 94
 Knuth, 95
 Largest progress, 32
 Lemke, 46
 Lewis, 258
 Lieberman, 47
 Liesegang, 98
 Linked list, 76
 - doubly, 79–80, 250
 - header, 77
 - implementation, 80
 - pointer, 77
 Liu, 85
 Loop unrolling, 64
 Lower bounds, 5
 LU decomposition, 135, 149, 313
 LU factorization, 122, 136, 158
 LU update, 153
 Main memory, 60, 62
 Maintainability, 61
 Major iteration, 186, 314
 Majthay, 29
 Markowitz number, 124–125, 141, 143
 Markowitz, 134, 136
 Maros, 60, 197, 199, 202, 204, 228, 232, 266, 277, 303, 310
 Matrix, 111
 - badly scaled, 111
 - lower triangular, 134
 - upper triangular, 134
 - well scaled, 111
 Measure of infeasibility, 204, 207, 216, 219
 Memory hierarchy, 62
 Memory scalability, 60
 Mészáros, 310
 MILP, xvii, 98, 274, 311
 Minor iteration, 186, 314
 Mitra, 98, 197, 202, 303
 Model editor, 310
 Model management system, 310
 Modeling languages, 310
 Modeling system, 314
 Modeling, 310
 Modularity, 61
 MPS file, 94
 MPS format, 94, 310
 MPS, 87
 Multitasking, 65
 Murtagh, 306
 Nazareth, 94, 232, 310
 NETLIB, 51
 Netlib/lp/data, 51
 Non-binding constraint, 7
 Nonnegativity restrictions, 4
 Nonzero pattern, 52, 54, 114, 252, 311
 Number of infeasibilities, 214
 Numerical stability, 140, 225, 241, 260, 291, 313
 Objective function, 4
 - composite, 227
 - dual, 38, 263, 268–269, 284
 - phase-1, 204, 232, 312
 - phase-2, 232
 - true, 227
 Optimality condition, 29, 166, 175–177, 261, 308–309
 Optimization algorithms, xviii
 Optimization software, 59, 311
 Optimizing compiler, 64
 Orchard-Hays, 5, xiii, 60, 111
 Orden, 230
 Osborne, 231, 234
 Padberg, 45
 Parameter file, 302
 Pattern matching, 95
 Perturbation, 178, 231, 241–243, 299
 PFI, 48, 122, 124, 128–129, 133, 136
 Phase-1, 37, 203, 214, 224–225, 244–245
 - dual, 277
 Phase-2, 37, 203, 234
 - dual, 277
 Piecewise linear, 207, 209, 268, 270, 281, 284
 Pipeline, 64
 Pivot column, 138
 Pivot element, 27, 169, 225
 Pivot position, 27
 Pivot row, 138, 281, 285
 Pivot step, 33
 Polyhedral set, 23–24
 Poole, 258
 Portability, 60, 67
 - binary, 60
 - source level, 60
 Postsolve, 97, 116
 Prékopa, xiii, 29
 Preprocessing, 97, 311
 Presolve history list, 118
 Presolve, 97–98, 117, 311
 PRICE, 33, 177, 304–305
 Pricing, 33, 177, 184
 - approximate steepest edge, 197, 207
 - Dantzig, 187, 193, 203, 285
 - dual, 293
 - Devex, 193, 195, 207
 - dual, 296
 - dual phase-1, 297
 - dual, 44, 285, 292
 - dynamic cyclic, 203
 - dynamic partial, 188
 - full, 185, 188, 203

- modified Devex, 196
- multiple, 186, 225, 313
- nested partial, 187
- normalized, 185, 189, 207, 285, 313
- partial, 185, 187
- sectional, 188, 203, 313
- steepest edge, 189–190, 195, 207
 - dual, 293
- Primal-dual pair, 38, 40, 261
- Problem, 38
 - degenerate, 230
 - dual, 38
 - large scale, 49, 204
 - primal, 38
- Product form of the inverse, 48
- Product form update, 135
- Profiling, 303
- PSM-1, 33–37, 161
- PSM-G, 161, 177
- RAM, 62–63
- Range constraint, 7
- Ratio test, 30, 173, 177–178, 225, 305
 - dual, 43–44, 264, 266, 286
 - Harris, 179, 181, 238–239, 243
 - phase-1, 217
 - extension, 217
 - primal, 30, 32–33
- Reconfigurable computing, 65
- Reduced cost, 28, 181, 189, 286
 - dual phase-1, 281, 286, 297, 312
 - dual phase-2, 312
 - phase-1, 205–206, 214–215, 219
- Redundant constraint, 102
- Refactorization, 314
- Reference framework, 193, 196
 - dual, 296
- Register, 62
- Reid, 85, 111–113, 140, 149, 190, 255
- Reinversion, 133, 314
- Relaxation, 41, 263, 274
- Representation theorem, 25
- Resende, 85
- Revised Simplex Method, 47
- Right-hand side, 4
- RISC, 63
- Robustness, 60
- Row basis, 307
- RSM, 47
- Run parameters, 301–302
 - algorithmic, 302–303
 - numerical, 302–303
 - string, 302
- Ryan, 231, 234
- Saunders, 149
- Scale factor, 110, 189
- Scaling, 97, 110, 117, 311
 - dynamic, 189–190
- Scatter, 71, 74
- Search direction, 30
- Sedgewick, 95
- Sense of the optimization, 303
- Sensitivity analysis, 314
- Shifting, 231, 241, 243–244, 299
- Simplex method, xvii, 19, 27
 - dual, 43, 240, 277, 291, 307, 311
 - primal, 28, 33, 161, 311
 - sparse, 69
 - tableau version, 47–48
 - two-phase, 37
- Simplex multiplier, 28, 177, 181, 185, 313
 - phase-1, 206
- SIMPRI, 200–203, 313
- Singleton column, 104–105
- Singleton row, 100
- Software engineering, xviii
- Software, 65
- Solution, 20, 162
 - basic feasible, 22, 25, 28, 163, 308
 - basic, 22, 28, 163
 - degenerate, 22, 164
 - feasible, 20, 162
 - infeasible basic, 214
 - infeasible, 35, 163
 - initial, 246
 - nondegenerate, 22, 164
 - optimal, 20, 25, 28, 44, 162
 - unbounded, 25, 28, 33, 41, 169, 172
- Sorting, 95, 226, 276, 290
 - bubble sort, 226
 - heap sort, 226
 - priority queue, 226, 287
 - selection sort, 226
- Sparsity, 50, 124, 140
- Spatial locality, 63
- Spike, 150–151
- SSX, 69–70, 133
- Stall, 64
- Stalling, 230, 234, 237
- Standard form, 3
- Steplength, 30, 180, 238–239, 306, 309
 - dual, 43, 265, 275, 288, 290
- Storage, 70
 - compressed, 70
 - compression, 76
 - explicit form, 70
 - full-length, 70, 74
 - ordered, 71
 - packed, 70
 - sparse matrix, 74
 - sparse vector, 70
- Strong duality, 39
- Structure, 52
- Sub-models, 310
- Suhl update, 152

- Suhl, 136, 141, 143, 149, 151
Sum of dual infeasibilities, 279, 281, 285, 289
Super sparsity, 50
Swietanowski, 197
Tearing algorithm, 255, 259
Temporal locality, 62
Threshold parameter, 126
Threshold pivoting, 126, 146
Tolerance, 110
 degeneracy, 231
 drop, 141
 feasibility, 179, 238
 expanding, 239
 initial, 240
 master, 238
 maximum, 240
 working, 238
optimality, 298
 absolute, 298
 relative, 298
Tolerances, 303
Tomlin, 98, 111, 149, 182
Triangularization, 126
Types of constraints, 11, 15
Types of variables, 11, 15
Unscaling, 117
Update, 34, 44, 177, 286
Upper bounds, 5
User friendliness, 61
Variable, 3
 artificial, 35–36
 basic, 21, 162
 blocking, 30
 bounded, 6
 decision, 3
 dominated, 106
 dual logical, 261–262, 264, 268, 286
 entering, 31
 fixed, 6
 free, 6, 283
 implied free, 104
 incoming, 31, 172, 176, 178, 204, 212, 224,
 281
 leaving, 31
 logical, 8, 11–12, 15–16, 18, 129, 135
 nonbasic, 21, 162
 nonnegative, 3, 6
 outgoing, 31, 172, 178, 204, 214, 280–281
 structural, 8, 11–12, 15–16, 246, 248, 260
 superbasic, 306
 unrestricted, 6
 weakly dominated, 106
Veiga, 85
Vertex, 23, 25, 307
 degenerate, 24, 230
Virtual perturbation, 234–235
Visual inspection, 311, 313
Weak duality, 39
Welch, 98
Williams, 98
Wolfe, 111, 230, 234
Wolfe's 'ad hoc' method, 231, 234, 238, 242,
 298