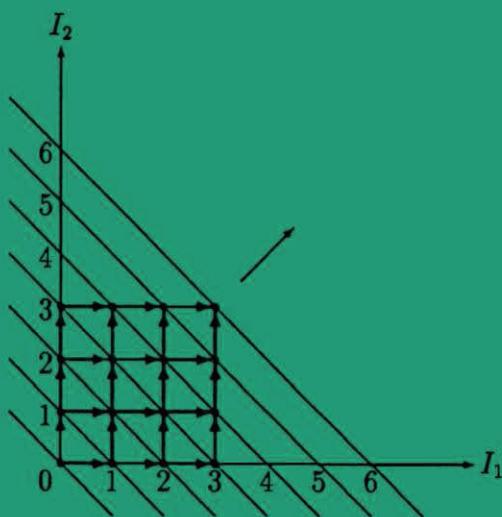


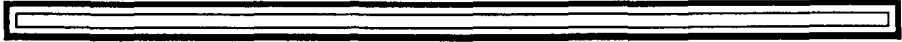
Loop Transformations for Restructuring Compilers

The Foundations

Utpal Banerjee



Kluwer Academic Publishers



Loop Transformations for Restructuring Compilers: The Foundations

Loop Transformations for Restructuring Compilers: The Foundations

By:

Utpal Banerjee
Intel Corporation



Kluwer Academic Publishers
Boston/Dordrecht/London

Distributors for North America:
Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA

Distributors for all other countries:
Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS

Library of Congress Cataloging-in-Publication Data

Banerjee, Utpal, 1942-

Loop transformations for restructuring compilers : the foundations / by
Utpal Banerjee.

p. cm.

Includes bibliographical references and index.

ISBN 0-7923-9318-X (alk. paper)

1. Compilers (Computer programs) 2. Loops (Group theory)
3. Parallel processing (Electronic computers) I. Title.

QA76.76.C65B36 1993

005.4'53'0151272--dc20

92-41647

CIP

Copyright © 1993 by Kluwer Academic Publishers

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photo-copying, recording, or otherwise, without the prior written permission of the publisher, Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061.

Printed on acid-free paper.

Printed in the United States of America

Contents

Preface	xiii
Acknowledgments	xvii
I Mathematical Background	1
1 Relations and Digraphs	3
1.1 Introduction	3
1.2 Relations	4
1.3 Lexicographic Order	11
1.4 Digraphs	15
2 Unimodular Matrices	21
2.1 Introduction	21
2.2 Basic Definitions	26
2.3 Echelon Reduction	32
2.4 Diagonalization	40
2.5 Permutation Matrices	46
3 Linear Equations and Inequalities	49
3.1 Introduction	49
3.2 Parts of a Number	52
3.3 Greatest Common Divisor	55
3.4 Diophantine Equations	59
3.5 Equations in Two Variables	66
3.6 Fourier's Method of Elimination	81

II Data Dependence	95
4 Basic Concepts	97
4.1 Introduction	97
4.2 Sequential Loop Nest	98
4.3 Dependence Definitions	103
5 Linear Dependence Problem	113
5.1 Introduction	113
5.2 Dependence Equation	119
5.3 Dependence Constraints	124
5.4 Dependence Algorithm	129
5.5 Regular Loop Nest	138
5.6 Rectangular Loop Nest	148
III Loop Transformations	161
6 Introduction to Loop Transformations	163
6.1 Introduction	163
6.2 Iteration Graph Partitioning	170
6.3 Unimodular Transformations	177
6.4 Loop Permutations	184
6.5 Loop Distribution	187
Appendix	195
Bibliography	293
Index	301

List of Figures

1.1	The poset of Example 1.2.	9
1.2	A digraph $G(V, E)$.	15
1.3	Acyclic digraphs defining the same partial order.	16
1.4	Condensation of Figure 1.2.	17
2.1	The polytope \mathcal{R} and its image \mathcal{RA} .	23
3.1	Graphs of the functions $i(t)$ and $j(t)$.	69
3.2	The points (i, j) in the solution-set Ψ .	71
3.3	Graphs of the functions $i(t)$ and $j(t)$.	77
3.4	The interval $[\tau_1, \tau_5] = [\tau_1, \tau_2] \cap (-\infty, \tau_3]$.	78
3.5	The interval $[\tau_6, \tau_2] = [\tau_1, \tau_2] \cap [\tau_4, \infty)$.	78
4.1	Index space of the loop nest in Example 4.1.	101
4.2	Statement dependence graph for Example 4.2.	107
4.3	Iteration dependence graph for Example 4.3.	109
4.4	Statement dependence graphs for Example 4.3.	110
6.1	Execution order of the program in Example 6.1.	165
6.2	A general program.	167
6.3	Execution order of the program in Figure 6.2.	168
6.4	Iteration dependence graph for Example 6.4.	173
6.5	Weakly connected components of Figure 6.4.	174
6.6	The digraph of Example 6.8 and its condensation.	188
6.7	Statement dependence graphs for Example 6.9.	191

List of Notations

In the following, $\mathbf{i} = (i_1, i_2, \dots, i_m)$ and $\mathbf{j} = (j_1, j_2, \dots, j_m)$ are two vectors of size m , and $1 \leq \ell \leq m$.

$S \times T$	Cartesian product of sets S and T	4
R^{-1}	Inverse of a relation R	4
\overline{R}	Transitive closure of R	4
\sim	A typical equivalence relation	5
$[a]$	Equivalence class of a (with respect to some equivalence relation)	5
\mathbf{Z}	Set of all integers	5
\leq	A typical partial order	6
$<$	Strict relation corresponding to \leq	6
(S, \leq)	A partially ordered set (poset)	6
$\wp(A)$	Power set of A	10
$a b$	Integer a (evenly) divides integer b	11
\mathbf{Z}^m	Set of all integer m -vectors	11
$\mathbf{0}$	Zero vector (size implied by context)	11
$\text{lev}(\mathbf{i})$	Level of a vector \mathbf{i}	11
$\text{sig}(i)$	Sign of an integer i	12
$\text{sig}(\mathbf{i})$	Sign of a vector \mathbf{i}	12
$\mathbf{i} \prec_\ell \mathbf{j}$	$i_1 = j_1, i_2 = j_2, \dots, i_{\ell-1} = j_{\ell-1}, i_\ell < j_\ell$	12
$\mathbf{i} \preceq_\ell \mathbf{j}$	$\mathbf{i} \prec_\ell \mathbf{j}$ or $\mathbf{i} = \mathbf{j}$	13
$\mathbf{i} \prec \mathbf{j}$	$\mathbf{i} \prec_\ell \mathbf{j}$ for some ℓ	13
$\mathbf{i} \preceq \mathbf{j}$	$\mathbf{i} \prec \mathbf{j}$ or $\mathbf{i} = \mathbf{j}$	13

$\mathbf{i} \succ_\ell \mathbf{j}$	$\mathbf{j} \prec_\ell \mathbf{i}$	13
$\mathbf{i} \succeq_\ell \mathbf{j}$	$\mathbf{j} \preceq_\ell \mathbf{i}$	13
$\mathbf{i} \succ \mathbf{j}$	$\mathbf{j} \prec \mathbf{i}$	13
$\mathbf{i} \succeq \mathbf{j}$	$\mathbf{j} \preceq \mathbf{i}$	13
$\mathbf{i} \succ \mathbf{0}$	\mathbf{i} is a positive vector (for some ℓ , $i_1 = i_2 = \dots = i_{\ell-1} = 0, i_\ell > 0$)	13
$\mathbf{i} \succeq \mathbf{0}$	\mathbf{i} is a nonnegative vector ($\mathbf{i} \succ \mathbf{0}$ or $\mathbf{i} = \mathbf{j}$)	13
$\mathbf{i} \prec \mathbf{0}$	\mathbf{i} is a negative vector (for some ℓ , $i_1 = i_2 = \dots = i_{\ell-1} = 0, i_\ell < 0$)	13
$\mathbf{i} \preceq \mathbf{0}$	\mathbf{i} is a nonpositive vector ($\mathbf{i} \prec \mathbf{0}$ or $\mathbf{i} = \mathbf{j}$)	13
$\mathbf{i} > \mathbf{j}$	$i_r > j_r$ for each r in $1 \leq r \leq m$	13
$\mathbf{i} \geq \mathbf{j}$	$i_r \geq j_r$ for each r in $1 \leq r \leq m$	13
$\mathbf{i} < \mathbf{j}$	$i_r < j_r$ for each r in $1 \leq r \leq m$	13
$\mathbf{i} \leq \mathbf{j}$	$i_r \leq j_r$ for each r in $1 \leq r \leq m$	13
$G(V, E)$	Digraph with vertex-set V and edge-set E	15
$\tilde{G}(\tilde{V}, \tilde{E})$	Condensation of digraph $G(V, E)$	17
$\overline{G}(V, \overline{E})$	Transitive closure of digraph $G(V, E)$	18
$\det(\mathbf{A})$	Determinant of a (square) matrix \mathbf{A}	21
\mathbf{A}^{-1}	Inverse of a nonsingular (square) matrix \mathbf{A}	21
\mathbf{R}^m	Set of all real m -vectors	22
$x \mapsto y$	A function that maps an element x of its domain to an element y of its range	22
$\text{col } j$	j^{th} column of a matrix.	26
\mathbf{A}'	Transpose of a matrix \mathbf{A}	26
\mathcal{I}_m	The $m \times m$ identity matrix	26
\mathcal{I}	An identity matrix (size implied by context).	26
$(\mathbf{A}; \mathbf{B})$	Column-augmented matrix (columns of \mathbf{B} added to columns of \mathbf{A})	26
(a_{ij})	Matrix whose element in row i and column j is a_{ij}	27

$\text{rank}(\mathbf{A})$	Rank of a matrix \mathbf{A}	27
a^+	$\max(a, 0)$ (positive part of a)	52
a^-	$\max(-a, 0)$ (negative part of a)	52
a/b	The exact result of division of a by b	55
$\gcd(a, b)$	Greatest common divisor of a and b	55
$S < T$	S precedes T in the body of loop nest	100
$\text{OUT}(S)$	Set of output variables of statement S	101
$\text{IN}(S)$	Set of input variables of statement S	101
$S \delta T$	Statement T depends on statement S	103
$S \bar{\delta} T$	T is indirectly dependent on S	104
$S \delta^f T$	T is flow dependent on S	105
$S \delta^a T$	T is anti-dependent on S	105
$S \delta^o T$	T is output dependent on S	105
$S \delta^i T$	T is input dependent on S	105
$(\mathbf{i}; \mathbf{j})$	Vector formed by concatenating components of \mathbf{i} with components of \mathbf{j}	116
$\begin{pmatrix} \mathbf{A} \\ \mathbf{B} \end{pmatrix}$	Row-augmented matrix (rows of \mathbf{B} added to rows of \mathbf{A})	116
$U_1 \diamond U_2$	$\{((i_1, i_2), (j_1, j_2)) : (i_r, j_r) \in U_r; r = 1, 2\}$	154

Preface

Automatic transformation of a sequential program into a parallel form is a subject that presents a great intellectual challenge, and at the same time promises a large practical reward. On one hand, there is a tremendous investment in existing sequential programs that we would like to run with ever increasing speed. On the other hand, scientists and engineers continue to write their application programs in sequential languages (primarily in Fortran), and keep demanding higher and higher speedups. Over a period of two decades, much research has been done on using restructuring compilers to enable the parallel execution of sequential programs. The original thrust came from Prof. David Kuck's group at the University of Illinois, and the Illinois work provided the starting point for Prof. Ken Kennedy's project at Rice. Research in this area continues at the Center for Supercomputing Research and Development (University of Illinois at Urbana-Champaign), at Rice University (Houston, Texas), and at several other places.

Instead of doing the unit computations of a program serially, we would like to do them simultaneously. However, there are two major obstacles: The integrity of the given program must be maintained, and the limitations of the given machine must not be exceeded. Preserving the integrity of the program means preserving its ‘dependence structure.’ This structure is created implicitly by the order in which the program—when executed sequentially—will read and write various memory locations. On the other hand, each parallel machine is limited by the number and type of operations it can perform simultaneously, and a variety of other constraints. The job of a restructuring compiler is to discover the dependence structure of

a given program and transform the program in a way that is consistent with both that dependence structure and the characteristics of the machine.

Much attention in this field of research has been focused on the Fortran **do** loop. This is where one expects to find major chunks of computation that need to be performed repeatedly for different values of the index variable. Many loop transformations have been designed over the years, and several of them can be found in any parallelizing compiler currently in use in industry or at a university research facility.

Our long term goal is to create a rigorous foundation for parallel processing work, much of which is inherently mathematical in nature. The present series deals with the transformation of a nest of Fortran-like loops. This volume provides the general mathematical background, discusses data dependence, and introduces some of the major transformations; while the next volume will build a complete theory of loop transformations based on the material developed here. The mathematical background part constitutes the foundation on which all other parts stand. It covers the *basic* mathematical concepts and tools necessary for an understanding of the subject; more advanced topics will be introduced later as needed. We have tried to provide advance motivation whenever possible, but sometimes the reader will encounter an idea or a theorem whose importance will only become clear later.

This book is directed towards graduate and upper division undergraduate students, and professional writers of restructuring compilers. A knowledge of programming languages and some knowledge of linear algebra are required. We define some of the standard linear algebra concepts (e.g., rank) in the text, while others are left undefined (e.g., linear dependence). No rigid set of rules was followed to decide which definitions to include and which ones to exclude. We also expect from the reader some familiarity with calculus and graph theory, and above all some mathematical maturity (whatever that might mean). No knowledge of diophantine equations or unimodular matrices is expected.

One of our goals is to build intuition into the mathematics of

loop transformations. Sometimes we deliberately leave out a step or two from an argument, and encourage the reader to supply the missing details. A missing step either turns up as a formal exercise or is simply flagged as a gap to be filled by a confidential comment to the reader (e.g., ‘proof?’).

We have tried to be precise and rigorous without appearing too pedantic. The examples and exercises are an essential part of the book; the reader is strongly urged to study the examples and do the exercises. Sometimes, a notion is introduced informally in an example or exercise long before it is given a formal definition. This way, the reader will already have an intuitive understanding of an important concept when its precise formulation is given.

Our main object of study is a nest of Fortran-like **do** loops, containing assignment and conditional statements involving scalars and subscripted variables. A loop limit is a linear (affine) function of the index variables of enclosing loops, and array subscripts are linear functions of the index variables. Loops may have arbitrary strides. In this volume, we deal with a simpler model of a perfect nest of loops (i.e., no statements between loops) where each loop has a stride of 1 and the body of the nest is a sequence of assignment statements. This model will be extended later to include imperfectly nested loops, conditional statements, and non-unit strides.

Number theory plays an important role in the theory of loop transformations. The index variable of a loop is an integer variable and array subscripts are integer valued functions. This leads naturally to mathematical objects like diophantine equations and unimodular matrices. We also have to deal with relations of various kinds. Dependence itself can be viewed as a relation among program variables, program statements, statement instances, or iterations of the nest. And relations, of course, lead to directed graphs. The main feature of the book is the display of loop transformations in a framework of these mathematical entities. Use of matrix notation has helped to simplify expressions and improve understanding of the main concepts. For a given transformation, the emphasis here is on showing what it is and when it is valid, rather than on discussing practical conditions under which it should be attempted.

The first part of the present volume contains one chapter each on relations and directed graphs, unimodular matrices, and diophantine equations and inequalities. The second part deals with data dependence. In Chapter 4, the basic dependence concepts are explained in terms of a perfect nest of loops with unit strides. In Chapter 5, the general linear dependence problem is formulated and its solution in two important special cases are discussed in detail. These two cases cover a large class of practical problems, and they can generate meaningful examples for loop transformations. Finer points of dependence analysis are beyond the scope of this book, although the reader should be able to get a clear picture of the fundamental concepts and a good understanding of the basic techniques. Finally, in Part III, Chapter 6 explains the concept of a valid loop transformation and introduces some of the major transformations via examples. The aim here is to demonstrate how the concepts and algorithms developed earlier can be used to decide the validity of a transformation and carry out its implementation. A complete theory of loop transformations will be presented in the next volume.

The programs in examples and exercises are written somewhat informally and are intended solely to illustrate the ideas in question in a minimum amount of space. For instance, there are no variable declarations and no distinction is explicitly made between integer and floating-point arithmetic. It is assumed that distinct symbols for arrays represent distinct arrays, array subscripts are always within the bounds, and that each value used in a program has been defined before the program is entered. Also, we have often used algebraic rather than programming notations (e.g., $X(2I_2)$).

The algorithms in the book are designed to be simple and direct, and they have been written to be easily implemented. In fact, an important part of the book is the Appendix where actual codes (in ANSI C) for most of the algorithms are given.

U.B.

Santa Clara, California

Acknowledgments

The author would like to thank the following people at Intel and the University of Illinois at Urbana-Champaign (UOI) for their help in the preparation of this book:

K. Sridharan (Intel) read the manuscript, pointed out errors, and wrote all the programs in the appendix. Those programs increase the usefulness of this book by a large extent.

David Sehr (Intel) read the manuscript more than once, found mistakes, gave many suggestions, and was always available for discussions. He did much of the reviewing work while preparing his PhD thesis at UOI, which adds another dimension to his effort.

Celso L. Mendes (UOI) read the manuscript very carefully and gave detailed comments. He has been reviewing the author's writings for many years.

Kevin Smith (Intel) read the same chapters many times and made valuable suggestions. He has probably suffered longer than any other reviewer on this particular project.

Beatrice Fu (Intel), and Jay Bharadwaj, Kent Fielden, Roland Kenner, Suresh Rao, Bill Savage and Pohua Chang of her compiler group read the manuscript and gave many valuable comments. They found time in their busy schedule to hold formal meetings and discuss the manuscript.

Sharad Mehrotra, Paul Petersen and Peng Tu of the Center for Supercomputing Research and Development (CSRD) at UOI read the manuscript, found errors, and gave suggestions. Prof. Constantine D. Polychronopoulos (CSRD) read part of the manuscript. Prof. David A. Padua, Associate Director of CSRD, helped with finding reviewers.

An earlier version of the manuscript was used as text for the course CS497, Section UB, at UOI in summer of 1991. That version was read by the students: H.-F. Chen, W. Y. Chen, Jr., G. E. Haab, M. E. Kendrat, R. Kumar, C.-F. Lim, S. A. Mahlke, S. Mehrotra, L.-S. Ong, J. W. Saputro, J. E. Sicolo, J. D. P. Womeldorf and K. Yi.

Nancy Navarro (Intel) took care of the proofreading and printing arrangements. She has given the author valuable help during the last days of writing. Kris Cavaco (Intel) did a good job of proofreading the manuscript in a short period of time. Dr. Dev Bose (Intel) gave advice about proofreading and printing.

Finally, the author wishes to thank Intel management and especially Dr. Richard Wirt for giving him the opportunity to write this book.



Loop Transformations for Restructuring Compilers: The Foundations

Part I

Mathematical Background

Chapter 1

Relations and Digraphs

1.1 Introduction

Relations and digraphs (directed graphs) are ubiquitous in the theory of loop transformations. To study something as basic as the execution ordering of the iterations of a loop nest, we need to know about several ways of partially ordering integer vectors. The most important concept in loop transformations, namely that of dependence, can be viewed as a relation between the members of a certain set. This set could be the set of iterations of a loop nest, or the set of statements in the body of the nest, or the set of instances of the statements. We can even go to a lower level and talk about dependence between ‘loads’ and ‘stores,’ or to a higher level and discuss dependence between large parts of a program.

The graphs that arise in the theory of loop transformations are digraphs for the most part, although it is occasionally necessary to consider the underlying undirected graph of a given directed graph. Digraphs are very closely associated with relations; in fact, in this book, we define a digraph simply as a nonempty set with a relation in it. The geometric point of view of digraphs is naturally suited to some situations and provides for a more intuitive description. For example, it is beneficial to represent the dependence relation between statements in a program as a digraph, and then use the terms and results from graph theory in the decomposition of the

program, based on dependence.

We assume that the reader is familiar with relations and both kinds of graphs. The purpose of this chapter is to collect those definitions and results in the theories of relations and digraphs that will be needed throughout the book. This chapter is, by necessity, rather condensed; it should not be used to learn about relations or digraphs for the first time. We have used the Halmos classic [Halm65] as the basis for the section on relations, and consulted [Even79], [Gibb85], [Knut73], and [Prat76] for the preparation of the section on digraphs.

1.2 Relations

The *Cartesian product*, $S \times T$, of two sets S and T is the set of all ordered pairs of the form (a, b) where $a \in S$ and $b \in T$. A *relation from* a set S to a set T is any subset of $S \times T$. A *relation in* a set S is a subset of $S \times S$. If R is a relation in S and $(a, b) \in R$, we write $a R b$. The *inverse* of a relation R , denoted by R^{-1} , is obtained by reversing each of the pairs belonging to R , so that $a R^{-1} b$ iff $b R a$. Let R^U denote the union and R^I the intersection of a collection of relations $\{R_k : k \in \mathcal{C}\}$ in S , where \mathcal{C} is some nonempty index set. Then $a R^U b$ iff $a R_k b$ for *some* k in \mathcal{C} , and $a R^I b$ iff $a R_k b$ for *each* k in \mathcal{C} . A relation R in a set S is

<i>reflexive</i>	if $a R a$ is true for each a ,
<i>irreflexive</i>	if $a R a$ is false for each a ,
<i>symmetric</i>	if $a R b$ implies $b R a$,
<i>antisymmetric</i>	if $a R b$ and $b R a$ imply $a = b$,
<i>transitive</i>	if $a R b$ and $b R c$ imply $a R c$,

where a , b , and c denote arbitrary elements of S . The *transitive closure*, \bar{R} , of a relation R in S is the smallest transitive relation in S containing R . We have $a \bar{R} b$ iff there is a sequence a_1, a_2, \dots, a_n of elements of S , such that

$$a = a_1, a_1 R a_2, a_2 R a_3, \dots, a_{n-1} R a_n, a_n = b.$$

A *partition* of a nonempty set S is a collection of pairwise disjoint nonempty subsets whose union is S . If two partitions $\{A_i\}$ and $\{B_j\}$ of the same set are such that each A_i is a subset of some B_j , then we say that the partition $\{A_i\}$ is *finer* than the partition $\{B_j\}$, and that $\{B_j\}$ is *coarser* than $\{A_i\}$.

A relation is an *equivalence relation* if it is reflexive, symmetric, and transitive. For an equivalence relation R in a set S , the *equivalence class* of a with respect to R is the set of all elements b in S such that $a R b$. When the relation R is understood, the equivalence class of a is denoted by $[a]$. The equivalence classes of an equivalence relation in S form a partition of S (Exercise 2). If an equivalence relation R_1 is contained in another equivalence relation R_2 (i.e., if $a R_1 b$ implies $a R_2 b$), then the partition formed by the equivalence classes with respect to R_1 is finer than the partition formed by the classes with respect to R_2 (Exercise 3).

Example 1.1 Define a relation \sim in \mathbf{Z} , the set of integers, by letting

$$a \sim b \text{ iff } (a - b) \bmod 5 = 0.$$

Then the following hold:

1. $a \sim a$ since 0 is a multiple of 5;
2. $a \sim b$ implies $b \sim a$, since if $a - b$ is a multiple of 5, then so is $b - a$;
3. $a \sim b$ and $b \sim c$ imply $a \sim c$, since if there are integers p and q such that $a - b = 5p$ and $b - c = 5q$, then $a - c = 5(p + q)$ where $(p + q)$ is an integer.

Thus, the relation \sim is an equivalence relation in \mathbf{Z} . The equivalence classes of $0, 1, 2, 3, 4$ are:

$$\begin{aligned}[0] &= \{0, 5, -5, 10, -10, \dots\} \\ [1] &= \{1, 6, -4, 11, -9, \dots\} \\ [2] &= \{2, 7, -3, 12, -8, \dots\} \\ [3] &= \{3, 8, -2, 13, -7, \dots\} \\ [4] &= \{4, 9, -1, 14, -6, \dots\}. \end{aligned}$$

Note that these five sets form a partition of \mathbf{Z} . The equivalence class of any integer with respect to \sim is one of these sets (proof?).

Suppose we define another relation in \mathbf{Z} by replacing ‘5’ with ‘10’ in the definition of \sim , and find the distinct equivalence classes with respect to this new relation. Those classes will also form a partition of \mathbf{Z} and it will be finer than the current partition (proof?).

A relation is a *partial order* if it is reflexive, antisymmetric, and transitive. A partial order is usually denoted by the symbol \leq . For every partial order \leq in a set S , an irreflexive and transitive relation $<$ can be defined as follows: For a and b in S , we have $a < b$ if $a \leq b$ and $a \neq b$. Conversely, from any irreflexive and transitive relation $<$ in S , a partial order \leq can be constructed by defining $a \leq b$ if either $a < b$ or $a = b$. We say that $<$ is the *strict relation* corresponding to \leq , and that \leq is the *weak relation* corresponding to $<$. The inverse of a partial order \leq is typically denoted by \geq , and the inverse of $<$ by $>$.

A *partially ordered set*, or *poset*, is a set together with a partial order in that set. Suppose (S, \leq) is a partially ordered set and a, b are elements of S . If $a < b$, then we say that a is a *predecessor* of b , and that b is a *successor* of a . If $a < b$ and there is no c in S such that $a < c < b$, then a is an *immediate predecessor* of b , and b is an *immediate successor* of a . The elements a and b are *comparable* if $a = b$, or $a < b$, or $b < a$. An element a is a *minimal element* if there is no x in S such that $x < a$, that is, if $a \leq x$ whenever x is comparable to a . An element a is the *least element* if $a \leq x$ for each x in S . The least element—in case it exists—is unique and the only minimal element. One can similarly define *maximal elements* and the *greatest element*. A subset T of S is a *chain* if two elements in T are always comparable; it is an *antichain* if two distinct elements in T are never comparable. The *length* of a chain (antichain) is the number of elements in it. A chain (antichain) is *maximal* if it is not a proper subset of any other chain (antichain). (A *proper* subset of a set S is a subset that is not equal to S .)

A *total order* in a set S is a partial order such that any two elements of S are always comparable, that is, the entire set S is one

chain. A *totally ordered set* is a set with a total order in it.

The maximal antichains of a poset form a partition of the set. We state below a result on this decomposition of partially ordered sets by L. Mirsky [Mirs71], in a form suitable for us:

Theorem 1.1 *Each nonempty, finite poset (S, \leq) has a partition (S_1, S_2, \dots, S_n) , where n is the length of the longest chain in S , such that*

- (a) *Each S_i is a maximal antichain ($1 \leq i \leq n$);*
- (b) *For $1 \leq i < j \leq n$, no element in S_i has a predecessor in S_j ;*
- (c) *For $1 < i \leq n$, every element in S_i has at least one immediate predecessor in S_{i-1} .*

PROOF. Each nonempty, finite poset has at least one minimal element (Exercise 11). Let S_1 denote the set of minimal elements of S , S_2 the set of minimal elements of $S - S_1$, and so on. These subsets satisfy the conditions (a)–(c) (Exercise 12). \square

Example 1.2 Let S denote the set of all ordered pairs of the form (I_1, I_2) where I_1 and I_2 are integers in $\{0, 1, 2, 3\}$ (see Figure 1.1). For (i_1, i_2) and (j_1, j_2) in S , define

$$(i_1, i_2) \leq (j_1, j_2) \quad \text{iff} \quad i_1 \leq j_1 \text{ and } i_2 \leq j_2.$$

For any elements (i_1, i_2) , (j_1, j_2) , and (k_1, k_2) of S , we have

1. $(i_1, i_2) \leq (i_1, i_2)$;
2. $(i_1, i_2) \leq (j_1, j_2)$ and $(j_1, j_2) \leq (i_1, i_2)$ imply $(i_1, i_2) = (j_1, j_2)$;
3. $(i_1, i_2) \leq (j_1, j_2)$ and $(j_1, j_2) \leq (k_1, k_2)$ imply $(i_1, i_2) \leq (k_1, k_2)$.

Hence, \leq is a partial order. Note that $(i_1, i_2) < (j_1, j_2)$ means that either $i_1 \leq j_1$ and $i_2 < j_2$, or $i_1 < j_1$ and $i_2 \leq j_2$ (i.e., the point (i_1, i_2) lies below and not to the right of the point (j_1, j_2) , or (i_1, i_2) lies to the left and not above (j_1, j_2)). The elements $(0, 0)$, $(1, 0)$, and $(0, 1)$ are all predecessors of the element $(1, 1)$, but only $(1, 0)$ and $(0, 1)$ are immediate predecessors. The only minimal element is the

least element $(0, 0)$, and the only maximal element is the greatest element $(3, 3)$. The partial order \leq is not a total order, since there are elements that are not comparable, for example, $(2, 3)$ and $(3, 2)$. The subset $\{(0, 0), (1, 1), (2, 2), (3, 3)\}$ is a chain, but not a maximal one, since it is properly contained in the chain

$$\{(0, 0), (0, 1), (1, 1), (1, 2), (2, 2), (2, 3), (3, 3)\}.$$

This longer chain is a maximal chain; its elements are shown as connected by a sequence of directed line segments in Figure 1.1.

The maximal antichains in (S, \leq) are the sets

$$\begin{aligned} S_1 &= \{(0, 0)\} \\ S_2 &= \{(1, 0), (0, 1)\} \\ S_3 &= \{(2, 0), (1, 1), (0, 2)\} \\ S_4 &= \{(3, 0), (2, 1), (1, 2), (0, 3)\} \\ S_5 &= \{(3, 1), (2, 2), (1, 3)\} \\ S_6 &= \{(3, 2), (2, 3)\} \\ S_7 &= \{(3, 3)\}. \end{aligned}$$

They form a partition of the set S . (In Figure 1.1, these antichains lie on the parallel lines with slope -1 .) This arrangement of the elements of S in a hierarchy of rows illustrates Theorem 1.1. The first row consists of the only minimal element of the set, which has no predecessors. The second row consists of all elements of S that are the minimal elements of $S - S_1$. Each member of the second row has an immediate predecessor, namely $(0, 0)$ of the first row. This process is continued; it stops after 7 rows, where 7 is the length of the longest chain in the original set S .

It is often necessary to embed a partial order in a total order. In general, there exist several total orders that include the partial order of a given poset. To see this, define an order in the poset (S, \leq) of Theorem 1.1 as follows: Choose arbitrary total orders in the subsets S_i , and stipulate that each element of a subset S_i will precede each element of a subset S_j whenever $i < j$. In other words, arrange the elements of the subset S_1 on a horizontal line in any order.

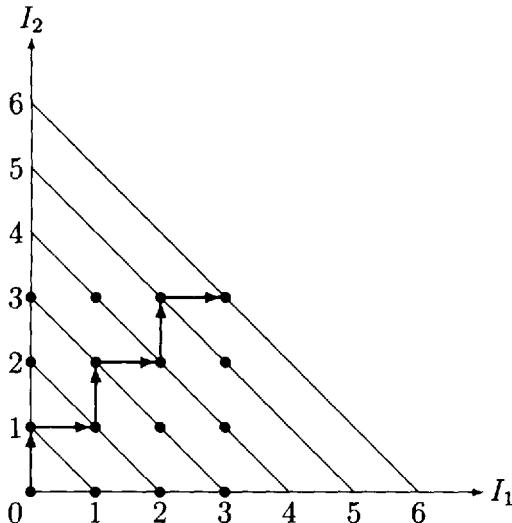


Figure 1.1: The poset of Example 1.2.

Then, put on the right of the rightmost element, the elements of the subset S_2 in any order, and so on. This arrangement is clearly a total order that includes the original partial order. There are variations of this method. The process of embedding a partial order in a total order is called *topological sorting*. For the description of a topological sorting algorithm, see Algorithm T in Section 2.2.3 of [Knut73]. This algorithm first outputs a minimal element x of the set S , removes that element from the set, then outputs a minimal element of the remainder $S - \{x\}$; and so on. The process stops when there are no elements left.

If we want the actual subsets S_1, S_2, \dots, S_n of Theorem 1.1, then Knuth's algorithm can be modified as follows: First, find all the minimal elements of S without any deletions, call that set S_1 ; delete S_1 from S ; find all the minimal elements of $S - S_1$ without any deletions, call that set S_2 ; delete S_2 from $S - S_1$; continue until an empty set is reached (Exercise 13).

EXERCISES 1.2

- Let $S = \{0, 1, 2, 3, 4, 5\}$. Find subsets of $S \times S$ that are relations (in S) of the following types: reflexive, irreflexive, symmetric, antisymmetric, transitive, equivalence, partial order, total order. Plot the points of $S \times S$

with respect to a pair of rectangular axes, and show those subsets on the plot.

2. Show that the equivalence classes with respect to an equivalence relation in a set form a partition of the set.
3. Show that if R_1 and R_2 are equivalence relations in a set and $R_1 \subset R_2$, then the partition formed by the equivalence classes with respect to R_1 is finer than the partition formed by the classes with respect to R_2 .
4. For a partition \mathcal{C} of a set S , define a relation R by requiring that $a R b$ iff a and b are two elements in the same member of \mathcal{C} . We call R the relation *induced* by \mathcal{C} . Prove that R is an equivalence relation. Now show that there is a one-to-one correspondence between the equivalence relations in a set and the partitions of the set.
5. In the definition of the relation \sim in Example 1.1, replace ‘5’ by any positive integer p , and denote the generalized relation by \sim_p . Show that \sim_p is an equivalence relation. Find the families of equivalence classes with respect to the relations \sim_2 and \sim_7 . Note that each family forms a partition of \mathbf{Z} .

Can you make any general statements about the partitions defined by two relations \sim_p and \sim_q , where p and q are any two positive integers? Give proofs and illustrations.

6. Let \mathbf{Z}^2 denote the set of all ordered pairs of integers. Define a relation \sim on \mathbf{Z}^2 by requiring that for two elements (i_1, i_2) and (j_1, j_2) , we have

$$(i_1, i_2) \sim (j_1, j_2) \quad \text{if} \quad (j_1 - i_1) \bmod 2 = 0 \quad \text{and} \quad (j_2 - i_2) \bmod 3 = 0.$$

Show that \sim is an equivalence relation in \mathbf{Z}^2 . How many equivalence classes are there? List them all.

7. Extend the previous exercise to \mathbf{Z}^m , the set of all m -tuples of integers, for any $m \geq 1$. Use arbitrary positive integers p_1, p_2, \dots, p_m for the mod operations.
8. Let A denote any set and $\wp(A)$ the *power set* of A , that is, the set of all subsets of A . Show that the relation of set inclusion \subset is a partial order in $\wp(A)$. Find a necessary and sufficient condition on A under which the poset $(\wp(A), \subset)$ is a totally ordered set.
9. Let S denote the set of all subsets of $\{1, 2, 3\}$. In the poset (S, \subset) , find the minimal and maximal elements, the least and the greatest elements, all chains and antichains. Mark the maximal chains and maximal antichains. Find the partition of S into its maximal antichains as described in Theorem 1.1.

Next, take for S the set of all nonempty subsets of $\{1, 2, 3\}$, and repeat the problem.

10. [Prat76] Let S denote the set of all positive integers less than or equal to 50. Show that the pair $(S, |)$ is a poset, where $a|b$ means that a (evenly) divides b . Find the minimal and maximal elements, the least and the greatest elements of this poset. Give examples of chains and antichains, maximal chains and maximal antichains. Find the partition of S into its maximal antichains as described in Theorem 1.1.
11. Show that a nonempty, finite poset has at least one minimal and one maximal element.
12. Complete the proof of Theorem 1.1.
13. Learn Algorithm T in Section 2.2.3 of [Knut73]. Modify it so that you can find the subsets S_1, S_2, \dots, S_n of Theorem 1.1. Test your algorithm on the posets of Exercise 9.

1.3 Lexicographic Order

The *Cartesian product*, $S_1 \times S_2 \times \dots \times S_m$, of m sets S_1, S_2, \dots, S_m is the set of all ordered m -tuples of the form (a_1, a_2, \dots, a_m) such that $a_1 \in S_1, a_2 \in S_2, \dots, a_m \in S_m$. If $S_1 = S_2 = \dots = S_m = S$, then a more convenient notation for the Cartesian product is S^m .

In this section, we consider the Cartesian product \mathbf{Z}^m where \mathbf{Z} is the set of all integers. The elements of \mathbf{Z}^m will be referred to as *integer vectors of size m* , or *integer m -vectors*, or *vectors in \mathbf{Z}^m* . A zero vector $(0, 0, \dots, 0)$ of any size is denoted by $\mathbf{0}$; the size of a given $\mathbf{0}$ will be clear from the context. The *leading element* of a nonzero vector $\mathbf{i} = (i_1, i_2, \dots, i_m)$ is its first nonzero element. If this leading element is i_ℓ , then the positive integer ℓ between 1 and m is called the *level* of \mathbf{i} , and is denoted by $\text{lev}(\mathbf{i})$. The *level* of the zero vector in \mathbf{Z}^m is defined to be $m + 1$. A vector \mathbf{i} is (*lexicographically*) *positive* or *negative* if its leading element is positive or negative, respectively.¹ Thus, the vectors $(1, -10)$, $(0, 2, -32)$ and $(0, 0, 2)$ are positive, and the vectors $(-1, 20)$, $(0, -2)$ and $(0, 0, -1)$ are negative. A *nonnegative vector* is either the zero vector or a positive vector.

¹By a ‘positive’ vector, we will always mean a vector that is lexicographically positive. A vector with all positive components is clearly a special kind of positive vector. Similar comments apply to negative vectors.

The sign of an integer i is denoted by $\text{sig}(i)$, where $\text{sig}(i) = 1$ if i is positive, $\text{sig}(i) = -1$ if i is negative, and $\text{sig}(0) = 0$. The *sign* of a vector $\mathbf{i} = (i_1, i_2, \dots, i_m)$ is the vector of signs of its components:

$$\mathbf{\text{sig}}(\mathbf{i}) = (\text{sig}(i_1), \text{sig}(i_2), \dots, \text{sig}(i_m)).$$

For example, the sign of $(-3, 8, 0)$ in \mathbf{Z}^3 is $(-1, 1, 0)$.

A vector \mathbf{i} is a *direction vector* if each of its components is one of the integers: $1, 0, -1$. Obviously, a given nonzero direction vector is the sign of infinitely many distinct vectors. It is useful to have a compact notation for a set of direction vectors that share a certain property or have a certain form. We will use the following shorthand for different subsets of $\{1, 0, -1\}$ with 2 or 3 elements:²

$$\begin{aligned} * &: \{1, 0, -1\} \\ \pm &: \{1, -1\} \\ 0+ &: \{0, 1\} \\ 0- &: \{0, -1\}. \end{aligned}$$

Using these symbols (and $1, 0, -1$) as components of a vector, we can construct various *direction vector forms* that represent sets of direction vectors. Thus, the notation $(0, 1, *)$ denotes the set of all positive direction vectors with level 2 in \mathbf{Z}^3 : $\{(0, 1, 1), (0, 1, 0), (0, 1, -1)\}$. We say that these vectors *have the form* $(0, 1, *)$. A given direction vector may have many different forms; for example, $(0, 1, -1)$ also has the form $(0, \pm, \pm)$. As another example, the direction vectors $(-1, 0, 0), (-1, 0, -1), (-1, 1, 0), (-1, 1, -1)$ all have the form $(-1, 0+, 0-)$.

For $1 \leq \ell \leq m$, we define a relation \prec_ℓ in \mathbf{Z}^m , by requiring that $\mathbf{i} \prec_\ell \mathbf{j}$ if

$$i_1 = j_1, i_2 = j_2, \dots, i_{\ell-1} = j_{\ell-1} \text{ and } i_\ell < j_\ell.$$

(See [Bane88].) In other words, $\mathbf{i} \prec_\ell \mathbf{j}$ iff the direction vector of $\mathbf{j} - \mathbf{i}$ is a positive vector of the form $(0, 0, \dots, 0, 1, *, *, \dots, *)$ with level

²Historically, the symbols $<, =, >$ have been used for direction vectors in place of $1, 0, -1$, respectively [Wolf82]. In that system, \neq stands for \pm , \leq for $0+$, and \geq for $0-$.

ℓ . For example, in \mathbf{Z}^3 , we have

$$\begin{aligned}(2, 15, 9) &\prec_1 (3, -2, 7) \\ (2, 15, 9) &\prec_2 (2, 16, -5) \\ (2, 15, 9) &\prec_3 (2, 15, 12).\end{aligned}$$

The notation $\mathbf{i} \preceq_\ell \mathbf{j}$ means either $\mathbf{i} \prec_\ell \mathbf{j}$ or $\mathbf{i} = \mathbf{j}$. It is easy to see that the relation \preceq_ℓ is a partial order (Exercise 2).

The *lexicographic order* \prec in \mathbf{Z}^m is defined to be the union of all the relations \prec_ℓ , that is,

$$\mathbf{i} \prec \mathbf{j} \quad \text{iff} \quad \mathbf{i} \prec_\ell \mathbf{j} \text{ for some } \ell \text{ in } 1 \leq \ell \leq m.$$

(See [Knut73].) The corresponding weak relation \preceq is a total order in \mathbf{Z}^m (Exercise 6). The inverses of the relations \prec_ℓ , \preceq_ℓ , \prec , and \preceq are denoted by \succ_ℓ , \succeq_ℓ , \succ , and \succeq , respectively. Note that a vector \mathbf{i} is positive if $\mathbf{i} \succ \mathbf{0}$, and negative if $\mathbf{i} \prec \mathbf{0}$. (We write $\mathbf{i} > \mathbf{0}$ if all components of \mathbf{i} are positive. The notations $\mathbf{i} \geq \mathbf{0}$, $\mathbf{i} < \mathbf{0}$, $\mathbf{i} \leq \mathbf{j}$, $\mathbf{i} > \mathbf{j}$, etc. have similar meanings.)

The proof of the following lemma is left to the reader:

Lemma 1.2 *Consider two vectors \mathbf{i} and \mathbf{j} in \mathbf{Z}^m . If $\mathbf{d} = \mathbf{j} - \mathbf{i}$ and $\boldsymbol{\sigma} = \text{sig}(\mathbf{d})$, then the following statements are equivalent:*

- (a) $\mathbf{i} \prec \mathbf{j}$,
- (b) $\mathbf{d} \succ \mathbf{0}$,
- (c) $\boldsymbol{\sigma} \succ \mathbf{0}$,
- (d) $\boldsymbol{\sigma}$ has one of the m forms:

$$(1, \underbrace{*}, \dots, \underbrace{*}), (0, 1, \underbrace{*}, \dots, \underbrace{*}), \dots, (\underbrace{0, \dots, 0}_{m-2}, 1, *), (\underbrace{0, \dots, 0}_{m-1}, 1).$$

Example 1.3 Consider the loop:

```
do I1 = 1, 100
  do I2 = 1, 200
    S :      X(I1, I2) = Y(I1, I2 + 1) + Z(I1 + 1, I2)
    enddo
  enddo
```

Let $S(i_1, i_2)$ denote the instance of the statement S when the index variables I_1 and I_2 have values i_1 and i_2 , respectively. Note that in the serial execution of the program, an instance $S(i_1, i_2)$ is executed before another instance $S(j_1, j_2)$ iff one of the following two conditions holds:

1. $i_1 < j_1$, that is, $(i_1, i_2) \prec_1 (j_1, j_2)$;
2. $i_1 = j_1$ and $i_2 < j_2$, that is, $(i_1, i_2) \prec_2 (j_1, j_2)$.

Using the lexicographic order, we can say that $S(i_1, i_2)$ is executed before $S(j_1, j_2)$ iff $(i_1, i_2) \prec (j_1, j_2)$. Compare the lexicographic order with the order defined in Example 1.2.

EXERCISES 1.3

1. Define a relation \sim in \mathbf{Z}^m , such that $\mathbf{i} \sim \mathbf{j}$ iff \mathbf{i} and \mathbf{j} have the same direction vector. Prove that \sim is an equivalence relation. For the case $m = 2$, show the equivalence classes on the $I_1 I_2$ -plane.
2. Show that for each ℓ in $1 \leq \ell \leq m$, the relation \preceq_ℓ is a partial order.
3. Show that given two vectors \mathbf{i} and \mathbf{j} in \mathbf{Z}^m , exactly one of the following is true: $\mathbf{i} = \mathbf{j}$, or $\mathbf{i} \prec_\ell \mathbf{j}$ for some ℓ , or $\mathbf{j} \prec_k \mathbf{i}$ for some k .
4. For given vectors \mathbf{i} and \mathbf{j} , can the inequality $\mathbf{i} \prec_\ell \mathbf{j}$ hold for more than one value of ℓ ? Explain.
5. Let $1 \leq p \leq m$ and $1 \leq q \leq m$. Given three vectors \mathbf{i} , \mathbf{j} , and \mathbf{k} in \mathbf{Z}^m , show that $\mathbf{i} \prec_p \mathbf{j}$ and $\mathbf{j} \prec_q \mathbf{k}$ together imply $\mathbf{i} \prec_\ell \mathbf{k}$, where $\ell = \min(p, q)$.
6. Prove that the relation \preceq in \mathbf{Z}^m is a total order.
7. Take the set S of Example 1.2. In the poset (S, \preceq_1) , find the minimal and maximal elements, the least and the greatest elements; give examples of chains and antichains. Mark the maximal chains and maximal antichains. Find the partition of S into its maximal antichains as described in Theorem 1.1.

Repeat the problem for the posets (S, \preceq_2) and (S, \preceq) .

8. On the $I_1 I_2$ -plane, show the region consisting of all elements $\mathbf{i} = (i_1, i_2)$ of \mathbf{Z}^2 such that (a) $\mathbf{i} \succ_1 \mathbf{0}$, (b) $\mathbf{i} \succ_2 \mathbf{0}$, (c) $\mathbf{i} \succ \mathbf{0}$. Let θ denote the radian measure of the angle $\angle POQ$, where O is the origin, and P and Q represent any two positive elements. Find the range of θ .

Extend the problem to \mathbf{Z}^3 and then solve it.

1.4 Digraphs

A *digraph* $G(V, E)$ consists of a nonempty set V and a relation E in V . Sometimes, it is convenient to say that G is *the graph of* the relation E . For $G(V, E)$, the *vertex-set* is V , a *vertex* is an element of V , the *edge-set* is E , and an *edge* is an element of E . (Thus, an edge is a unique ordered pair of vertices.) A digraph is *finite* if its vertex-set and edge-set are both finite. We will work with finite digraphs only. It helps to visualize a digraph as a geometrical figure where a vertex u is represented by a point, and an edge (u, v) by a line-segment directed from the point u to the point v .

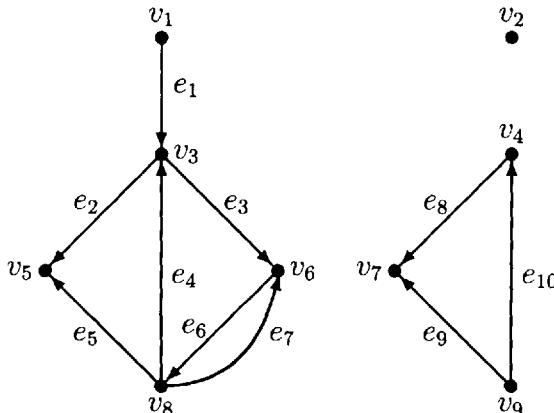


Figure 1.2: A digraph $G(V, E)$.

Let $e = (u, v)$ denote an edge of a digraph $G(V, E)$. Then e is said to be *directed from* u to v . The *end points* of e are u and v , the *initial vertex* of e is u , and its *final vertex* is v . An edge is a *self-loop* if its initial and final vertices are the same. If u and v are two distinct vertices, then an edge directed from u to v and an edge directed from v to u are said to be *antiparallel*. (Note that there are no ‘parallel’ edges, since any two edges that have the same initial vertex and the same final vertex must be identical.) A vertex is said to be *isolated* if there is no edge directed from it and no edge directed to it. In the digraph of Figure 1.2, the two edges e_6 and e_7 between vertices v_6 and v_8 are antiparallel, and the only isolated

vertex is v_2 .

For $n \geq 1$, a (*directed*) *path* of length n from a vertex u to a vertex v is a sequence of (not necessarily distinct) vertices (v_0, v_1, \dots, v_n) such that $v_0 = u$, $v_n = v$, and (v_{k-1}, v_k) is an edge for each k in $1 \leq k \leq n$. A *cycle* is a path (v_0, v_1, \dots, v_n) such that $v_n = v_0$. A digraph is *acyclic* if it has no cycles. In Figure 1.2, (v_1, v_3, v_6, v_8) is a path of length 3 from v_1 to v_8 , and (v_3, v_6, v_8, v_3) and $(v_3, v_6, v_8, v_6, v_8, v_3)$ are cycles.

An acyclic digraph $G(V, E)$ defines a partial order in V in a natural way: For two elements u and v of V , define $u \leq v$ if $u = v$, or there is a path from u to v in G . (Note that there cannot be a path from a vertex u to itself.) The relation \leq is reflexive, transitive, and antisymmetric (proof?), and therefore it is a partial order. Since we are dealing with paths, two acyclic digraphs with the same vertex-set V , but with different edge-sets, can represent the same partial order in V . An example of this is seen in Figure 1.3, where each graph defines the following partial order: $v_1 \leq v_2$, $v_1 \leq v_3$, $v_1 \leq v_4$, $v_2 \leq v_4$, $v_3 \leq v_4$. (See Exercise 2.)

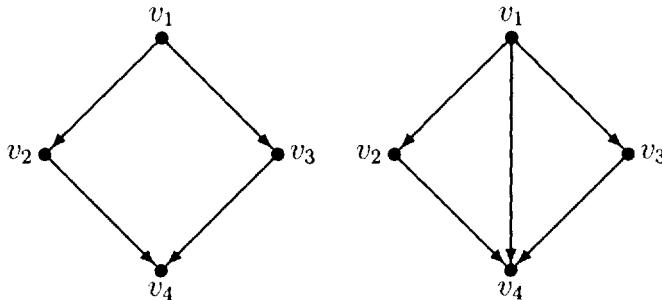


Figure 1.3: Acyclic digraphs defining the same partial order.

Two distinct vertices u and v of a digraph $G(V, E)$ are *strongly connected* if there is a path in G from u to v , and a path from v to u . The digraph itself is *strongly connected* if any two distinct vertices in it are always strongly connected. If $G(V, E)$ is not strongly connected, it can be broken up into its ‘strongly connected parts.’ Define a relation \sim in V such that for u, v in V , we have $u \sim v$ if $u = v$, or u and v are strongly connected. It is easy to see that \sim is

an equivalence relation in V . The equivalence classes with respect to this relation are called the *strongly connected components* of the digraph G . Thus, the strongly connected components of G form a partition of V . The *condensation* of $G(V, E)$ is a digraph $\tilde{G}(\tilde{V}, \tilde{E})$ obtained from $G(V, E)$ as follows:

1. Replace each strongly connected component of G by a vertex in \tilde{V} .
2. For an ordered pair of distinct strongly connected components (C_1, C_2) , if the set $\{(u, v) \in E : u \in C_1, v \in C_2\}$ is nonempty, then replace it by a single edge in \tilde{E} from C_1 to C_2 .

Given a digraph G , we can find its condensation $\tilde{G}(\tilde{V}, \tilde{E})$ by the well-known Tarjan algorithm ([Tarj72], [Even79], [Gibb85]). The condensation of a digraph is always acyclic (Exercise 3), and hence there is a natural partial order in the set of strongly connected components. The condensation of the digraph in Figure 1.2 is shown in Figure 1.4, where

$$C_1 = \{v_1\}, \quad C_2 = \{v_2\}, \quad C_3 = \{v_3, v_6, v_8\}, \quad C_4 = \{v_4\}, \\ C_5 = \{v_5\}, \quad C_6 = \{v_7\}, \quad C_7 = \{v_9\}.$$

The digraph in Figure 1.4 defines the following partial order between the strongly connected components of G : $C_1 \leq C_3 \leq C_5$, $C_4 \leq C_6$, $C_7 \leq C_4$, $C_7 \leq C_6$.

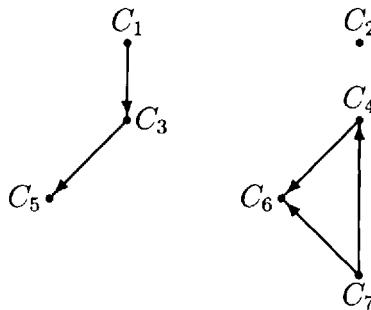


Figure 1.4: Condensation of Figure 1.2.

A weaker notion of connectedness can be defined in a digraph if we ignore the directions of the edges. For $n \geq 1$, an *undirected path*

of *length* n joining two vertices u and v is a sequence of vertices (v_0, v_1, \dots, v_n) such that $v_0 = u$, $v_n = v$, and there is an edge in E with end points v_{k-1} and v_k for each k in $1 \leq k \leq n$. Two distinct vertices are *weakly connected* if there is an undirected path joining them. The digraph itself is *weakly connected* if any two distinct vertices in it are always weakly connected. As in the case of strong connectedness, we can define an equivalence relation in V such that two vertices are related if they are identical or weakly connected. The equivalence classes of this relation are called the *weakly connected components* of the digraph $G(V, E)$. Thus, the weakly connected components form another partition of the vertex-set V . There are three weakly connected components of the digraph in Figure 1.2: $\{v_1, v_3, v_5, v_6, v_8\}$, $\{v_2\}$ and $\{v_4, v_7, v_9\}$.

The *transitive closure* of a digraph $G(V, E)$ is the digraph $\overline{G}(V, \overline{E})$ that has V as the vertex-set and the transitive closure of E as the edge-set. Thus, $\overline{G}(V, \overline{E})$ is constructed from G by adding an edge from a vertex u to a vertex v whenever there is a path, but no edge, from u to v in G . The second digraph in Figure 1.3 is the transitive closure of the first digraph.

A digraph $G'(V', E')$ is a *subgraph* of a digraph $G(V, E)$, if V' is included in V and E' in E . A subgraph $G'(V', E')$ is said to be *induced* by V' , if E' contains all edges in E that have both end points in V' . The *union* of two digraphs $G_1(V, E_1)$ and $G_2(V, E_2)$ (with the same vertex-set V) is the digraph $G(V, E_1 \cup E_2)$ obtained by taking the union of the two edge-sets.

EXERCISES 1.4

1. Draw a digraph $G(V, E)$ with three or more vertices, where the relation E in V is
 - (a) The empty relation
 - (b) A reflexive relation
 - (c) An irreflexive relation
 - (d) A transitive relation
 - (e) A relation that is not transitive
 - (f) An irreflexive and transitive relation
 - (g) A partial order
 - (h) An equivalence relation.

2. (a) Show that if \leq is a partial order in a nonempty set V , then the digraph $G(V, \leq)$ of the corresponding strict relation is acyclic.
- (b) If $G(V, E)$ is an acyclic digraph, then E is clearly an irreflexive relation in V . Give an example of an acyclic digraph $G(V, E)$ where E is *not* transitive.
- (c) Prove that if $G(V, E_1)$ and $G(V, E_2)$ are two acyclic digraphs with the same transitive closure, then they define the same partial order in V .
3. Prove that the condensation \tilde{G} of a digraph G is acyclic. How is \tilde{G} related to G when G itself is acyclic?
4. Can you always make a cycle out of the vertices in a strongly connected component of a digraph? Do the vertices on a cycle always form a strongly connected component? Give proofs and/or examples.
5. Show that the partition of the vertex-set of a digraph into its strongly connected components is finer than the partition into its weakly connected components.
6. Show that for a vertex v in a digraph $G(V, E)$, the singleton set $\{v\}$ is a strongly connected component of G , if any one of the following holds:
 - (a) v is a source (i.e., there is no edge directed to v);
 - (b) v is a sink (i.e., there is no edge directed from v);
 - (c) v is an isolated vertex (both a source and a sink).
 If $\{v\}$ is a strongly connected component, then must v be a source, or a sink, or an isolated vertex? Give examples.
7. Give an example where $G(V, E)$ is a digraph, V' is a subset of V , and E' a subset of E , but the pair (V', E') does not define a digraph. (All sets are to be nonempty.)
8. Learn the Tarjan Algorithm from one of the references cited in the text or from any other source. Use it to find the condensation of the digraph $G(V, E)$, where V is the set $\{1, 2, 3, 4, 5, 6\}$ and E is the relation:
 - (a) $\{(1, 1), (1, 5), (2, 3), (3, 4), (4, 2), (4, 5), (5, 1)\}$
 - (b) $\{(1, 2), (2, 3), (4, 5), (5, 6), (6, 1)\}$
 - (c) $\{(1, 1), (2, 2), (3, 3), (4, 4)\}$
 - (d) $\{(1, 3), (2, 1), (3, 2), (3, 4), (4, 5), (4, 6), (5, 4), (6, 5)\}$.

Chapter 2

Unimodular Matrices

2.1 Introduction

A matrix is *integral* or an *integer matrix*, if its elements are all integers. A *rational* or *real* matrix is defined in the same way. These definitions are also extended to vectors.

Since the index variables of the loops in a program are integer-valued, in analyzing loops, we frequently need *integer* solutions to equations, and *integer* matrices to transform a set of *integer* vectors to another set of *integer* vectors. Equations in integer variables are called *diophantine equations*; they are discussed in Chapter 3. The present chapter gives brief descriptions of some topics in the theory of integer matrices. Consult any linear algebra text for a review of matrices, and see [Schr87] for a detailed discussion on unimodular matrices and related topics.

The class of integer matrices is closed under many standard matrix operations, with one notable exception: The inverse of a non-singular integer matrix is not always integral. A unimodular matrix is defined in such a way that its inverse is necessarily integral. We say that a square integer matrix \mathbf{A} is *unimodular* if $\det(\mathbf{A}) = \pm 1$. This definition guarantees that the inverse \mathbf{A}^{-1} exists, is an integer matrix, and is itself unimodular. Unimodular matrices arise naturally in the solution of diophantine equations (Chapter 3). They are also highly desirable as transformation instruments, since such

a matrix \mathbf{A} represents a one-to-one mapping, and has the property that $\mathbf{x}\mathbf{A}$ is an integer vector iff \mathbf{x} is an integer vector (of the right size).

Let \mathbf{R}^m denote the Cartesian product of m copies of the real line, $m \geq 1$. The elements of \mathbf{R}^m are the real m -vectors (vectors of size m). An (*affine*) *half-space* in \mathbf{R}^m is a set of the form

$$\{(x_1, x_2, \dots, x_m) \in \mathbf{R}^m : a_1x_1 + a_2x_2 + \dots + a_mx_m \leq c\}$$

for some nonzero vector (a_1, a_2, \dots, a_m) and some real number c . A *polytope* in \mathbf{R}^m is a bounded subset that is the intersection of a finite number of half-spaces.

An $m \times m$ nonsingular real matrix \mathbf{A} defines the one-to-one mapping $\mathbf{x} \mapsto \mathbf{x}\mathbf{A}$ of \mathbf{R}^m onto itself. The image of a polytope \mathcal{R} under this mapping is another polytope $\mathcal{R}\mathbf{A}$. If \mathbf{A} is integral, then distinct integer vectors of \mathcal{R} are mapped into distinct integer vectors of $\mathcal{R}\mathbf{A}$. If \mathbf{A} is also unimodular, then each integer vector in $\mathcal{R}\mathbf{A}$ is also the image of some integer vector in \mathcal{R} . In other words, a unimodular matrix defines a one-to-one mapping of the integer vectors of \mathcal{R} onto the integer vectors of $\mathcal{R}\mathbf{A}$.

The iterations of a typical loop nest seen in practice are defined by the set of integer vectors in a certain polytope. Transforming those vectors by a unimodular matrix, we create a new program that has the same iterations, but a new execution order. It is often possible to choose a matrix such that the transformed program is equivalent to the original program, and the new order of iterations leads to a faster execution of the program on a given architecture. Matrix transformations of a loop nest are discussed in the following example:

Example 2.1 In the loop nest

```

 $L_1:$       do  $I_1 = 0, 10$ 
 $L_2:$           do  $I_2 = 0, I_1 + 5$ 
                   $H(I_1, I_2)$ 
                  enddo
              enddo
  
```

the index variables I_1 and I_2 satisfy the inequalities:

$$\begin{aligned} 0 &\leq I_1 \leq 10 \\ 0 &\leq I_2 \leq I_1 + 5. \end{aligned}$$

The set of values of (I_1, I_2) consists of all the integer vectors in the following subset of \mathbf{R}^2 :

$$\mathcal{R} = \{(x_1, x_2) \in \mathbf{R}^2 : 0 \leq x_1 \leq 10, 0 \leq x_2 \leq x_1 + 5\}. \quad (2.1)$$

The region \mathcal{R} is clearly bounded, and it is the intersection of the half-spaces

$$-x_1 \leq 0, x_1 \leq 10, -x_2 \leq 0, -x_1 + x_2 \leq 5,$$

so that \mathcal{R} is a polytope in \mathbf{R}^2 . We will contrast the transformations of \mathcal{R} by two nonsingular integer matrices, one unimodular and the other not.

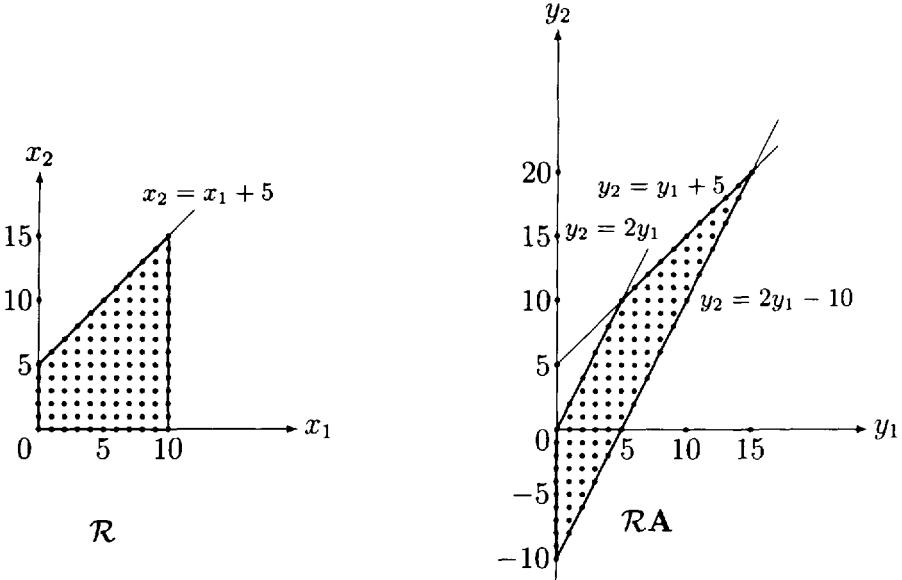


Figure 2.1: The polytope \mathcal{R} and its image \mathcal{RA} .

First, we study the transformation $\mathbf{x} \mapsto \mathbf{x}\mathbf{A}$ of \mathcal{R} under a unimodular matrix \mathbf{A} . Let

$$\mathbf{A} = \begin{pmatrix} 0 & -1 \\ 1 & 2 \end{pmatrix} \text{ so that } \mathbf{A}^{-1} = \begin{pmatrix} 2 & 1 \\ -1 & 0 \end{pmatrix}.$$

A vector (x_1, x_2) is mapped into a vector

$$(y_1, y_2) = (x_1, x_2)\mathbf{A} = (x_2, -x_1 + 2x_2),$$

while each vector (y_1, y_2) is the image of the vector

$$(x_1, x_2) = (y_1, y_2)\mathbf{A}^{-1} = (2y_1 - y_2, y_1).$$

We rewrite the conditions in (2.1) defining \mathcal{R} in terms of y_1 and y_2 :

$$\begin{aligned} 0 &\leq 2y_1 - y_2 \leq 10 \\ 0 &\leq y_1 \leq 2y_1 - y_2 + 5, \end{aligned}$$

and find that (Exercise 2)

$$\mathcal{R}\mathbf{A} = \{(y_1, y_2) : 0 \leq y_1 \leq 15, 2y_1 - 10 \leq y_2 \leq \min(2y_1, y_1 + 5)\}.$$

The polytopes \mathcal{R} and $\mathcal{R}\mathbf{A}$ are shown in Figure 2.1.

The iterations of the loop nest (L_1, L_2) are labeled using the integer vectors in \mathcal{R} . Since there is a one-to-one correspondence between the integer vectors in \mathcal{R} and the integer vectors in $\mathcal{R}\mathbf{A}$, we can relabel the iterations of the loop nest (L_1, L_2) using the integer vectors in $\mathcal{R}\mathbf{A}$. Thus, the following loop nest has exactly the same iterations as (L_1, L_2) (but arranged in a different order¹):

```

do  $K_1 = 0, 15$ 
  do  $K_2 = 2K_1 - 10, \min(2K_1, K_1 + 5)$ 
     $H(2K_1 - K_2, K_1)$ 
  enddo
enddo

```

Next, we take a matrix that is nonsingular, but not unimodular.

Let

$$\mathbf{B} = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \text{ so that } \mathbf{B}^{-1} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix}.$$

The connection between the old variables x_1, x_2 and the new variables y_1, y_2 is given by $(y_1, y_2) = (2x_1, x_2)$. The set \mathcal{RB} can be put in the form (Exercise 2):

$$\mathcal{RB} = \{(y_1, y_2) : 0 \leq y_1 \leq 20, 0 \leq y_2 \leq y_1/2 + 5\}.$$

¹Whether or not the new loop nest is logically equivalent to the old loop nest is another matter. We will return to the equivalence question in Section 6.3.

Distinct integer vectors in \mathcal{R} have been mapped into distinct integer vectors in \mathcal{RB} . However, there are more integer vectors in \mathcal{RB} than there are in \mathcal{R} , since some real vectors in \mathcal{R} are mapped into integer vectors in \mathcal{RB} . For example, $(y_1, y_2) = (7, 3)$ is the image of $(x_1, x_2) = (\frac{7}{2}, 3)$. We see that an integer vector (y_1, y_2) in \mathcal{RB} is the image of an integer vector in \mathcal{R} iff y_1 is even, so that the loop nest

```
do  $K_1 = 0, 20, 2$ 
  do  $K_2 = 0, K_1/2 + 5, 1$ 
     $H(K_1/2, K_2)$ 
  enddo
enddo
```

has the same iterations as (L_1, L_2) .

In general, it is more difficult to represent the transformation of a loop nest by a non-unimodular matrix than the transformation by a unimodular matrix.

Each unimodular matrix is the result of a finite sequence of ‘elementary’ row (or column) operations performed on the identity matrix of the same size. By using unimodular matrices, we can reduce an arbitrary matrix to a special form (‘echelon’ or ‘diagonal’). These reductions are important for solving diophantine equations and transforming loops; they are studied in detail in the following sections. We start by giving some basic definitions in Section 2.2; then discuss the echelon and diagonalization algorithms in sections 2.3 and 2.4, respectively; and finally take a brief look at the permutation matrices—a special kind of unimodular matrices—in Section 2.5. Properties of unimodular matrices are developed along the way.

EXERCISES 2.1

1. Prove that
 - (a) The transpose of a unimodular matrix is unimodular;
 - (b) The inverse of a unimodular matrix is unimodular;
 - (c) The product of two unimodular matrices is unimodular.

2. Prove that the forms of $\mathcal{R}\mathbf{A}$ and $\mathcal{R}\mathbf{B}$ given in Example 2.1 are indeed correct.

2.2 Basic Definitions

In this section, we state some of the elementary definitions of matrix theory, customizing them for integer matrices, as needed. The material presented here is standard for the most part and can be found in any linear algebra book, except that we have introduced a couple of new terms for easy reference. From now on, a matrix is an integer matrix, unless explicitly designated to be otherwise. The rows of an $m \times n$ matrix are labeled $1, 2, \dots, m$ from top to bottom, and the columns $1, 2, \dots, n$ from left to right. Thus, for example, row 4 is ‘lower’ than row 3, and col 6 is on the right of col 5. (Sometimes we use the abbreviation ‘col’ for *column*.)

In expressions involving matrices and vectors, it is always tacitly assumed that their sizes are compatible. For example, if we write a matrix product \mathbf{AB} , it is understood that the number of columns of \mathbf{A} is equal to the number of rows of \mathbf{B} . We do not distinguish between a row vector and a matrix with only one row, or between a column vector and a matrix with only one column. The notation (x_1, x_2, \dots, x_m) will be used for both row and column vectors. For emphasis, a ‘·’ may sometimes be used to indicate a matrix product, or the inner product of two vectors. The transpose of a matrix \mathbf{A} is denoted by \mathbf{A}' . The $m \times m$ identity matrix is denoted by \mathcal{I}_m , or simply by \mathcal{I} if the size is understood.

A *submatrix* of a given matrix \mathbf{A} is the matrix obtained by deleting some rows and columns of \mathbf{A} . A *subdeterminant* of \mathbf{A} is the determinant of a square submatrix of \mathbf{A} . If \mathbf{A} is an $m \times p$ matrix and \mathbf{B} an $m \times q$ matrix, then the *column-augmented matrix* $(\mathbf{A}; \mathbf{B})$ is the $m \times (p + q)$ matrix whose first p columns are the columns of \mathbf{A} and the last q columns are the columns of \mathbf{B} . Thus,

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 0 & 0 \\ 0 & 5 \end{pmatrix} \text{ and } \mathbf{B} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \text{ yield } (\mathbf{A}; \mathbf{B}) = \left(\begin{array}{cc|c} 1 & 2 & 1 \\ 0 & 0 & 0 \\ 0 & 5 & 0 \end{array} \right).$$

(The vertical line is used to separate the elements of \mathbf{A} from the elements of \mathbf{B} .) A *row-augmented matrix* is defined similarly. Note that $\mathbf{C} \cdot (\mathbf{A}; \mathbf{B}) = (\mathbf{CA}; \mathbf{CB})$ for any $r \times m$ matrix \mathbf{C} .

A matrix (a_{ij}) is *upper triangular* if $a_{ij} = 0$ whenever $i > j$, *lower triangular* if $a_{ij} = 0$ whenever $i < j$, and a *diagonal matrix* if $a_{ij} = 0$ whenever $i \neq j$. The (*main*) *diagonal* of an $m \times n$ matrix (a_{ij}) is the vector $(a_{11}, a_{22}, \dots, a_{pp})$ where $p = \min(m, n)$. The *rank* of a matrix \mathbf{A} , denoted by $\text{rank}(\mathbf{A})$, is the maximum number of linearly independent rows of \mathbf{A} . Note that $\text{rank}(\mathbf{A})$ is also the maximum number of linearly independent columns of \mathbf{A} .

For a given $m \times n$ matrix \mathbf{A} , let ℓ_i denote the column number of the leading element of row i . (For a zero row, ℓ_i is undefined.) Then, \mathbf{A} is an *echelon matrix* if for some integer ρ in $0 \leq \rho \leq m$, the following conditions hold:

1. The rows 1 through ρ are nonzero rows;
2. The rows $\rho + 1$ through m are zero rows;
3. For $1 \leq i \leq \rho$, each element in column ℓ_i below row i is zero;
4. $\ell_1 < \ell_2 < \dots < \ell_\rho$.

(The leading element of a row need not be equal to 1.) A zero matrix is an echelon matrix for which $\rho = 0$. Three nonzero echelon matrices with the values 4, 3, 2 of ρ , respectively, are given below:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}, \quad \begin{pmatrix} 5 & 2 & 3 \\ 0 & -1 & 0 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0 & 5 & 2 & 3 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Since the nonzero rows of an echelon matrix \mathbf{A} are linearly independent, it follows that this integer ρ is equal to $\text{rank}(\mathbf{A})$.

Any matrix can be ‘reduced’ to an echelon matrix and also to a diagonal matrix. We will explain in the next two sections what ‘reduction’ means in each case and how it can be accomplished. A reduction algorithm consists of a sequence of elementary row and/or column operations performed on one or more matrices. The three types of *elementary row operations* for an integer matrix are:²

²These names are not standard.

1. *reversal*: Multiply a row by -1 ;
2. *interchange*: Interchange two rows;
3. *skewing*: Add an integer multiple of one row to another row.

The *elementary column operations* are defined similarly.

An *elementary matrix* is any matrix obtained from an identity matrix by one elementary row operation. The matrix and the row operation are said to *correspond* to each other. Thus, there are three types of elementary matrices. An elementary matrix is a *reversal matrix*, an *interchange matrix*, or a *skewing matrix*, if the corresponding row operation is a reversal, interchange, or skewing, respectively. A skewing matrix is an *upper (lower) skewing matrix*, if the corresponding row operation adds a multiple of a row to a higher (lower) row. In other words, an upper (lower) skewing matrix is obtained from the identity matrix by replacing a zero above (below) the main diagonal by an integer.

The elementary matrices can also be derived by column operations. For example, the same matrix is obtained from the 3×3 identity matrix, if we interchange rows 1 and 3, or interchange columns 1 and 3. The elementary matrix obtained by a column reversal, column interchange, or column skewing is a reversal, interchange, or skewing matrix, respectively (as defined above in terms of row operations). An upper (lower) skewing matrix is obtained by adding a multiple of a column to a column on its right (left). An elementary matrix and the column operation that produces it are said to *correspond* to each other.

Example 2.2 In the 2×2 size, there are two reversal matrices:

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix};$$

one interchange matrix:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix};$$

and one set of upper and one set of lower skewing matrices:

$$\begin{pmatrix} 1 & z \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 \\ z & 1 \end{pmatrix},$$

respectively, where z denotes any integer.

Similarly, there are three 3×3 reversal matrices:

$$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix};$$

three interchange matrices:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \text{ and } \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix};$$

three sets of upper skewing matrices:

$$\begin{pmatrix} 1 & z & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & z \\ 0 & 0 & 1 \end{pmatrix};$$

and three sets of lower skewing matrices:

$$\begin{pmatrix} 1 & 0 & 0 \\ z & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ z & 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & z & 1 \end{pmatrix}.$$

In the product \mathbf{AB} of two matrices \mathbf{A} and \mathbf{B} , we say that \mathbf{B} is *premultiplied* by \mathbf{A} , and that \mathbf{A} is *postmultiplied* by \mathbf{B} . An elementary (row or column) operation on a matrix \mathbf{A} can be realized by forming the product between \mathbf{A} and the corresponding elementary matrix in the proper order. This is formally stated in the following lemma whose proof we omit.

Lemma 2.1 *On a given matrix \mathbf{A} , performing an elementary row operation is equivalent to premultiplying \mathbf{A} by the corresponding elementary matrix, and performing an elementary column operation is equivalent to postmultiplying \mathbf{A} by the corresponding elementary matrix.*

Example 2.3 Consider any 3×2 matrix

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}.$$

The three 3×3 elementary matrices

$$\mathbf{E}_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \mathbf{E}_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \text{ and } \mathbf{E}_3 = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

are of type reversal, interchange, and skewing, respectively. We form the products

$$\mathbf{E}_1 \mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ -a_{21} & -a_{22} \\ a_{31} & a_{32} \end{pmatrix}, \mathbf{E}_2 \mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \\ a_{21} & a_{22} \end{pmatrix},$$

and

$$\mathbf{E}_3 \mathbf{A} = \begin{pmatrix} a_{11} + 2a_{31} & a_{12} + 2a_{32} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$$

to multiply the second row of \mathbf{A} by -1 , interchange rows 2 and 3, and add 2 times row 3 to row 1, respectively. These are precisely the row operations that correspond to the three elementary matrices.

The 2×2 elementary matrices

$$\mathbf{F}_1 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \mathbf{F}_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \text{ and } \mathbf{F}_3 = \begin{pmatrix} 1 & 3 \\ 0 & 1 \end{pmatrix}$$

are of type reversal, interchange, and skewing, respectively. Forming the products

$$\mathbf{A} \mathbf{F}_1 = \begin{pmatrix} -a_{11} & a_{12} \\ -a_{21} & a_{22} \\ -a_{31} & a_{32} \end{pmatrix}, \mathbf{A} \mathbf{F}_2 = \begin{pmatrix} a_{12} & a_{11} \\ a_{22} & a_{21} \\ a_{32} & a_{31} \end{pmatrix},$$

and

$$\mathbf{AF}_3 = \begin{pmatrix} a_{11} & a_{12} + 3a_{11} \\ a_{21} & a_{22} + 3a_{21} \\ a_{31} & a_{32} + 3a_{31} \end{pmatrix},$$

we accomplish the following column operations on \mathbf{A} : multiply the first column by -1 , interchange the two columns, and add 3 times the first column to the second column, respectively. Note that these are the column operations that correspond to $\mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3$.

All elementary matrices are unimodular. The determinant of a reversal or interchange matrix is -1 , and the determinant of a skewing matrix is $+1$. Note that an elementary row or column operation does not change the rank of a matrix (Exercise 3).

Since a product of unimodular matrices is again unimodular, the following result is a direct consequence of Lemma 2.1:

Lemma 2.2 *Applying a finite sequence of elementary row operations to an $m \times n$ matrix \mathbf{A} is equivalent to premultiplying \mathbf{A} by a suitable $m \times m$ unimodular matrix. Applying a finite sequence of elementary column operations to \mathbf{A} is equivalent to postmultiplying \mathbf{A} by a suitable $n \times n$ unimodular matrix.*

EXERCISES 2.2

- Under what condition is a square diagonal matrix unimodular?
- Prove that the absolute value of the determinant of a square matrix remains unchanged under any elementary row or column operation on the matrix.
- Show that the rank of an $m \times m$ elementary matrix is m . Show also that for a given matrix \mathbf{A} , $\text{rank}(\mathbf{EA}) = \text{rank}(\mathbf{AF}) = \text{rank}(\mathbf{A})$ for any compatible elementary matrices \mathbf{E} and \mathbf{F} .
- Show that the transpose and the inverse of an elementary matrix are elementary matrices. Write down the transpose and the inverse of each 2×2 elementary matrix.
- Compare the row and column operations corresponding to an elementary matrix with those corresponding to its transpose and inverse. Give examples using 2×2 and 3×3 matrices.

6. Show how the elementary row and column operations on a 2×2 matrix can be accomplished by forming products with suitable elementary matrices.
7. Repeat the previous problem with a 3×5 matrix.
8. Prove that a lower (upper) skewing matrix \mathbf{A} can be expressed as $\mathbf{A} = \mathbf{T}_1 \mathbf{B} \mathbf{T}_2$ where \mathbf{B} is an upper (lower) skewing matrix and \mathbf{T}_1 and \mathbf{T}_2 are interchange matrices.
9. Prove that if a matrix is unimodular, then it remains unimodular after any sequence of elementary row and column operations. Then, prove the converse: If the result of a sequence of elementary row and column operations to a matrix \mathbf{A} is a unimodular matrix, then \mathbf{A} itself is unimodular.

2.3 Echelon Reduction

Reducing an $m \times n$ matrix \mathbf{A} to *echelon form* means finding an $m \times m$ unimodular matrix \mathbf{U} and an $m \times n$ echelon matrix \mathbf{S} , such that $\mathbf{UA} = \mathbf{S}$. Such reduction is always possible. The matrices \mathbf{U} and \mathbf{S} are not unique; we describe here a method that finds one set of such matrices.

Any integer matrix can be reduced to echelon form by a finite sequence of elementary row operations. To get an idea of this process, assume that the first column (written horizontally) of the given matrix \mathbf{A} is $(10, 5, 7, 3, 0)$. We start with rows 3 and 4, since row 4 is the lowest row in which the first column has a nonzero element. The greatest multiple of 3 that goes into 7 is $6 = 3 \times 2$. We subtract 2 times row 4 from row 3 in \mathbf{A} , so that the first column becomes $(10, 5, 1, 3, 0)$. (We are not tracking the other columns, but they also change.) An interchange of rows 3 and 4 in \mathbf{A} turns the first column into $(10, 5, 3, 1, 0)$. We subtract 3 times row 4 from row 3 in \mathbf{A} , and then interchange rows 3 and 4 again. This leaves us with the first column $(10, 5, 1, 0, 0)$. The matrix \mathbf{A} is a little closer to an echelon matrix than when we started. Next, we deal with rows 2 and 3. This process continues until the first column gets the form $(z, 0, 0, 0, 0)$ where z is a nonzero integer. Column 2 of \mathbf{A} is worked on next until it has the form $(x, y, 0, 0, 0)$. We stop when \mathbf{A} becomes an echelon matrix.

Define $\mathbf{U} = \mathcal{I}_m$ and $\mathbf{S} = \mathbf{A}$. Then, \mathbf{U} is unimodular and the relation $\mathbf{U}\mathbf{A} = \mathbf{S}$ holds. We apply a sequence of steps changing both matrices \mathbf{U} and \mathbf{S} , with the goal of turning \mathbf{S} into an echelon matrix, while keeping \mathbf{U} unimodular and the identity $\mathbf{U}\mathbf{A} = \mathbf{S}$ unchanged.

When a row operation is applied to \mathbf{U} and \mathbf{S} , they change into the matrices $\mathbf{E}\mathbf{U}$ and $\mathbf{E}\mathbf{S}$, respectively, where \mathbf{E} is the elementary matrix corresponding to that operation (Lemma 2.1). After assigning to \mathbf{U} and \mathbf{S} their new values, we see that the equation $\mathbf{U}\mathbf{A} = \mathbf{S}$ remains intact, and that the new \mathbf{U} is also unimodular (Exercise 2.1.1(c)). We also note that applying the same row operation to the two matrices \mathbf{U} and \mathbf{S} separately is equivalent to applying that operation to the column-augmented matrix $(\mathbf{U}; \mathbf{S})$.

When \mathbf{S} becomes an echelon matrix, its final value and the final value of \mathbf{U} provide the echelon reduction of the original matrix \mathbf{A} .

Algorithm 2.1 (Echelon Reduction) Given an $m \times n$ integer matrix \mathbf{A} , this algorithm finds an $m \times m$ unimodular matrix \mathbf{U} and an $m \times n$ echelon matrix $\mathbf{S} = (s_{ij})$, such that $\mathbf{U}\mathbf{A} = \mathbf{S}$. We start with $\mathbf{U} = \mathcal{I}_m$, $\mathbf{S} = \mathbf{A}$, and work on the matrix $(\mathbf{U}; \mathbf{S})$. Let i_0 denote the row number in which the last processed column of \mathbf{S} had a nonzero element.

```

set  $\mathbf{U} \leftarrow \mathcal{I}_m$ ,  $\mathbf{S} \leftarrow \mathbf{A}$ ,  $i_0 \leftarrow 0$ 
do  $j = 1, n, 1$ 
  if there is at least one nonzero  $s_{ij}$  with  $i_0 < i \leq m$ 
    then
      set  $i_0 \leftarrow i_0 + 1$ 
      do  $i = m, i_0 + 1, -1$ 
        do while  $s_{ij} \neq 0$ 
          set  $\sigma \leftarrow \text{sig}(s_{(i-1)j} * s_{ij})$ 
           $z \leftarrow \lfloor |s_{(i-1)j}| / |s_{ij}| \rfloor$ 
          subtract  $\sigma z$ (row  $i$ ) from row  $(i-1)$  in  $(\mathbf{U}; \mathbf{S})$ 
          interchange rows  $i$  and  $(i-1)$  in  $(\mathbf{U}; \mathbf{S})$ 
        enddo
      enddo
    endif
  enddo

```

□

Example 2.4 We will apply Algorithm 2.1 to reduce the 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} 4 & 4 & 1 \\ 6 & 0 & 1 \\ 4 & 3 & 2 \end{pmatrix}$$

to echelon form. Start with the 3×6 column-augmented matrix

$$(\mathbf{U}; \mathbf{S}) \equiv (\mathcal{I}_3; \mathbf{A}) = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 4 & 4 & 1 \\ 0 & 1 & 0 & 6 & 0 & 1 \\ 0 & 0 & 1 & 4 & 3 & 2 \end{array} \right),$$

and perform the following sequence of operations. The variables j , i_0 , i , σ , and z have the same meanings as in the algorithm.

$$\begin{aligned} j = 1, i_0 = 1, i = 3, & \quad \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 4 & 4 & 1 \\ 0 & 1 & -1 & 2 & -3 & -1 \\ 0 & 0 & 1 & 4 & 3 & 2 \end{array} \right). \\ \sigma = 1, z = \lfloor 6/4 \rfloor = 1. & \\ \text{Subtract row 3 from row 2:} & \end{aligned}$$

$$\begin{aligned} \text{Interchange rows 2 and 3:} & \quad \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 4 & 4 & 1 \\ 0 & 0 & 1 & 4 & 3 & 2 \\ 0 & 1 & -1 & 2 & -3 & -1 \end{array} \right). \end{aligned}$$

$$\begin{aligned} j = 1, i_0 = 1, i = 3, & \quad \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 4 & 4 & 1 \\ 0 & -2 & 3 & 0 & 9 & 4 \\ 0 & 1 & -1 & 2 & -3 & -1 \end{array} \right). \\ \sigma = 1, z = \lfloor 4/2 \rfloor = 2. & \\ \text{Subtract 2(row 3) from row 2:} & \end{aligned}$$

$$\begin{aligned} \text{Interchange rows 2 and 3:} & \quad \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 4 & 4 & 1 \\ 0 & 1 & -1 & 2 & -3 & -1 \\ 0 & -2 & 3 & 0 & 9 & 4 \end{array} \right). \end{aligned}$$

$$\begin{aligned} j = 1, i_0 = 1, i = 2, & \quad \left(\begin{array}{ccc|ccc} 1 & -2 & 2 & 0 & 10 & 3 \\ 0 & 1 & -1 & 2 & -3 & -1 \\ 0 & -2 & 3 & 0 & 9 & 4 \end{array} \right). \\ \sigma = 1, z = \lfloor 4/2 \rfloor = 2. & \\ \text{Subtract 2(row 2) from row 1:} & \end{aligned}$$

$$\begin{aligned} \text{Interchange rows 1 and 2:} & \quad \left(\begin{array}{ccc|ccc} 0 & 1 & -1 & 2 & -3 & -1 \\ 1 & -2 & 2 & 0 & 10 & 3 \\ 0 & -2 & 3 & 0 & 9 & 4 \end{array} \right). \end{aligned}$$

$$j = 2, i_0 = 2, i = 3,$$

$$\sigma = 1, z = \lfloor 10/9 \rfloor = 1.$$

Subtract row 3 from row 2:

$$\left(\begin{array}{ccc|ccc} 0 & 1 & -1 & 2 & -3 & -1 \\ 1 & 0 & -1 & 0 & 1 & -1 \\ 0 & -2 & 3 & 0 & 9 & 4 \end{array} \right).$$

Interchange rows 2 and 3:

$$\left(\begin{array}{ccc|ccc} 0 & 1 & -1 & 2 & -3 & -1 \\ 0 & -2 & 3 & 0 & 9 & 4 \\ 1 & 0 & -1 & 0 & 1 & -1 \end{array} \right).$$

$$j = 2, i_0 = 2, i = 3,$$

$$\sigma = 1, z = \lfloor 9/1 \rfloor = 9.$$

Subtract 9(row 3) from row 2:

$$\left(\begin{array}{ccc|ccc} 0 & 1 & -1 & 2 & -3 & -1 \\ -9 & -2 & 12 & 0 & 0 & 13 \\ 1 & 0 & -1 & 0 & 1 & -1 \end{array} \right).$$

Interchange rows 2 and 3:

$$\left(\begin{array}{ccc|ccc} 0 & 1 & -1 & 2 & -3 & -1 \\ 1 & 0 & -1 & 0 & 1 & -1 \\ -9 & -2 & 12 & 0 & 0 & 13 \end{array} \right).$$

The algorithm stops at this point and we get

$$\mathbf{U} = \left(\begin{array}{ccc} 0 & 1 & -1 \\ 1 & 0 & -1 \\ -9 & -2 & 12 \end{array} \right) \text{ and } \mathbf{S} = \left(\begin{array}{ccc} 2 & -3 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 13 \end{array} \right).$$

In some applications, for a given matrix \mathbf{A} , we need to find a unimodular matrix \mathbf{V} and an echelon matrix \mathbf{S} , such that $\mathbf{A} = \mathbf{VS}$. It is clear that this \mathbf{V} could be the inverse of the unimodular matrix \mathbf{U} of Algorithm 2.1. We can, of course, use that algorithm first and then compute \mathbf{U}^{-1} . However, if \mathbf{U} is not needed at all, Algorithm 2.1 can be modified to yield directly a \mathbf{V} and an \mathbf{S} such that $\mathbf{A} = \mathbf{VS}$.

Define $\mathbf{V} = \mathcal{I}_m$ and $\mathbf{S} = \mathbf{A}$. Then, \mathbf{V} is unimodular and the relation $\mathbf{A} = \mathbf{VS}$ holds. We apply a sequence of steps changing both matrices \mathbf{V} and \mathbf{S} , with the goal of turning \mathbf{S} into an echelon matrix, while keeping \mathbf{V} unimodular and the identity $\mathbf{A} = \mathbf{VS}$ unchanged.

When a row operation is applied to \mathbf{S} , it changes into the matrix \mathbf{ES} , where \mathbf{E} is the elementary matrix corresponding to that operation (Lemma 2.1). If we assign to \mathbf{S} this new value \mathbf{ES} , and to \mathbf{V}

the value $\mathbf{V}\mathbf{E}^{-1}$, then \mathbf{V} will remain unimodular, and the relation $\mathbf{A} = \mathbf{VS}$ will still hold. Whenever a row operation (represented by an elementary matrix \mathbf{E}) is applied to \mathbf{S} , we apply the corresponding column operation (represented by \mathbf{E}^{-1}) to \mathbf{V} . (See Exercise 2.2.5.) If the row operation is interchanging rows r and s , then the column operation is interchanging columns r and s . If the row operation is adding $k(\text{row } s)$ to row r , then the column operation is subtracting $k(\text{col } r)$ from col s .

When \mathbf{S} becomes an echelon matrix, its final value and the final value of \mathbf{V} provide the modified echelon reduction of the original matrix \mathbf{A} .

Algorithm 2.2 (Modified Echelon Reduction) Given an $m \times n$ integer matrix \mathbf{A} , this algorithm finds an $m \times m$ unimodular matrix \mathbf{V} and an $m \times n$ echelon matrix $\mathbf{S} = (s_{ij})$, such that $\mathbf{A} = \mathbf{VS}$. We start with $\mathbf{V} = \mathcal{I}_m$, $\mathbf{S} = \mathbf{A}$, and work on the matrices \mathbf{V} and \mathbf{S} . Let i_0 denote the row number in which the last processed column of \mathbf{S} had a nonzero element.

```

set  $\mathbf{V} \leftarrow \mathcal{I}_m$ ,  $\mathbf{S} \leftarrow \mathbf{A}$ ,  $i_0 \leftarrow 0$ 
do  $j = 1, n, 1$ 
  if there is at least one nonzero  $s_{ij}$  with  $i_0 < i \leq m$ 
    then
      set  $i_0 \leftarrow i_0 + 1$ 
      do  $i = m, i_0 + 1, -1$ 
        do while  $s_{ij} \neq 0$ 
          set  $\sigma \leftarrow \text{sig}(s_{(i-1)j} * s_{ij})$ 
           $z \leftarrow \lfloor |s_{(i-1)j}| / |s_{ij}| \rfloor$ 
          subtract  $\sigma z$ (row  $i$ ) from row  $(i-1)$  in  $\mathbf{S}$ 
          add  $\sigma z$ (col  $(i-1)$ ) to col  $i$  in  $\mathbf{V}$ 
          interchange rows  $i$  and  $(i-1)$  in  $\mathbf{S}$ 
          interchange columns  $i$  and  $(i-1)$  in  $\mathbf{V}$ 
        enddo
      enddo
    endif
  enddo

```

□

Note that in each algorithm, $\text{rank}(\mathbf{A}) = \text{rank}(\mathbf{S})$ (why?). Also, we can extend either algorithm such that the diagonal elements of \mathbf{S} are nonnegative (explain).

Example 2.5 For the matrix of Example 2.4:

$$\mathbf{A} = \begin{pmatrix} 4 & 4 & 1 \\ 6 & 0 & 1 \\ 4 & 3 & 2 \end{pmatrix},$$

we will find a unimodular matrix \mathbf{V} and an echelon matrix \mathbf{S} , such that $\mathbf{A} = \mathbf{VS}$. Start with $\mathbf{S} = \mathbf{A}$ and $\mathbf{V} = \mathcal{I}_3$. The variables j , i_0 , i , σ , and z have the same meanings as in Algorithm 2.2.

$$j = 1, i_0 = 1, i = 3, \sigma = 1, z = \lfloor 6/4 \rfloor = 1.$$

Subtract row 3 from row 2 in \mathbf{S} ; add col 2 to col 3 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 4 & 4 & 1 \\ 2 & -3 & -1 \\ 4 & 3 & 2 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

Interchange rows 2 and 3 in \mathbf{S} , and columns 2 and 3 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 4 & 4 & 1 \\ 4 & 3 & 2 \\ 2 & -3 & -1 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

$$j = 1, i_0 = 1, i = 3, \sigma = 1, z = \lfloor 4/2 \rfloor = 2.$$

Subtract 2(row 3) from row 2 in \mathbf{S} ; add 2(col 2) to col 3 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 4 & 4 & 1 \\ 0 & 9 & 4 \\ 2 & -3 & -1 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 3 \\ 0 & 1 & 2 \end{pmatrix}.$$

Interchange rows 2 and 3 in \mathbf{S} , and columns 2 and 3 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 4 & 4 & 1 \\ 2 & -3 & -1 \\ 0 & 9 & 4 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 1 \\ 0 & 2 & 1 \end{pmatrix}.$$

$$j = 1, i_0 = 1, i = 2, \sigma = 1, z = \lfloor 4/2 \rfloor = 2.$$

Subtract 2(row 2) from row 1 in \mathbf{S} ; add 2(col 1) to col 2 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 0 & 10 & 3 \\ 2 & -3 & -1 \\ 0 & 9 & 4 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 1 & 2 & 0 \\ 0 & 3 & 1 \\ 0 & 2 & 1 \end{pmatrix}.$$

Interchange rows 1 and 2 in \mathbf{S} , and columns 1 and 2 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 2 & -3 & -1 \\ 0 & 10 & 3 \\ 0 & 9 & 4 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 2 & 1 & 0 \\ 3 & 0 & 1 \\ 2 & 0 & 1 \end{pmatrix}.$$

$$j = 2, i_0 = 2, i = 3, \sigma = 1, z = \lfloor 10/9 \rfloor = 1.$$

Subtract row 3 from row 2 in \mathbf{S} ; add col 2 to col 3 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 2 & -3 & -1 \\ 0 & 1 & -1 \\ 0 & 9 & 4 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 2 & 1 & 1 \\ 3 & 0 & 1 \\ 2 & 0 & 1 \end{pmatrix}.$$

Interchange rows 2 and 3 in \mathbf{S} , and columns 2 and 3 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 2 & -3 & -1 \\ 0 & 9 & 4 \\ 0 & 1 & -1 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 2 & 1 & 1 \\ 3 & 1 & 0 \\ 2 & 1 & 0 \end{pmatrix}.$$

$$j = 2, i_0 = 2, i = 3, \sigma = 1, z = \lfloor 9/1 \rfloor = 9.$$

Subtract 9(row 3) from row 2 in \mathbf{S} ; add 9(col 2) to col 3 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 2 & -3 & -1 \\ 0 & 0 & 13 \\ 0 & 1 & -1 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 2 & 1 & 10 \\ 3 & 1 & 9 \\ 2 & 1 & 9 \end{pmatrix}.$$

Interchange rows 2 and 3 in \mathbf{S} , and columns 2 and 3 in \mathbf{V} :

$$\mathbf{S} \leftarrow \begin{pmatrix} 2 & -3 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 13 \end{pmatrix}, \mathbf{V} \leftarrow \begin{pmatrix} 2 & 10 & 1 \\ 3 & 9 & 1 \\ 2 & 9 & 1 \end{pmatrix}.$$

The algorithm stops at this point.

EXERCISES 2.3

- Algorithm 2.1 shows that for any given matrix \mathbf{A} , there exists a pair of matrices (\mathbf{U}, \mathbf{S}) such that \mathbf{U} is unimodular, \mathbf{S} is echelon, and $\mathbf{U}\mathbf{A} = \mathbf{S}$. Explain how you can construct a different pair of matrices $(\mathbf{U}_1, \mathbf{S}_1)$ with the same properties. How many possible choices are there in general?
- In Example 2.4, write down the 10 elementary matrices that correspond to the 10 elementary row operations. Show—by actual computation—that the product of these matrices (taken in the proper order) equals \mathbf{U} , and that $\mathbf{U}\mathbf{A}$ is indeed equal to \mathbf{S} .
- Apply algorithm 2.1 to each of the following matrices:

$$(a) \begin{pmatrix} 1 & 0 & 0 \\ 2 & 2 & -1 \\ 1 & -1 & 1 \end{pmatrix}$$

$$(b) \begin{pmatrix} 5 & 4 & -2 \\ -10 & -13 & 8 \\ -12 & -20 & 13 \end{pmatrix}$$

$$(c) \begin{pmatrix} 1 & 2 & 2 \\ -6 & 8 & 5 \\ 6 & -6 & -3 \end{pmatrix}$$

$$(d) \begin{pmatrix} 3 & 1 & -2 & 4 \\ 1 & 0 & 2 & 3 \\ 2 & 1 & -1 & 1 \end{pmatrix}$$

$$(e) \begin{pmatrix} 1 & -2 & 3 & -1 \\ 2 & -1 & 2 & 2 \\ 3 & 1 & 2 & 3 \end{pmatrix}.$$

- Apply Algorithm 2.2 to the matrices of the previous exercise.

2.4 Diagonalization

Diagonalization (or *reduction to diagonal form*) of an $m \times n$ matrix \mathbf{A} consists of finding an $m \times m$ unimodular matrix \mathbf{U} , an $n \times n$ unimodular matrix \mathbf{V} , and an $m \times n$ diagonal matrix \mathbf{D} , such that $\mathbf{UAV} = \mathbf{D}$. Diagonalization of a matrix is always possible. The matrices \mathbf{U} , \mathbf{V} , \mathbf{D} are not unique; we describe here a method that finds one set of such matrices.

Any integer matrix can be reduced to a diagonal matrix by a finite sequence of elementary row *and* column operations. The diagonal matrix \mathbf{D} of our algorithm will have the form

$$\begin{pmatrix} d_1 & & & & \\ & d_2 & & & \\ & & \ddots & & \\ & & & d_\rho & \\ & & & & 0 \\ & & & & & \ddots & \\ & & & & & & 0 \end{pmatrix}$$

where the diagonal elements d_1, d_2, \dots, d_ρ are nonzero integers. If needed, we can always extend our algorithm such that d_1, d_2, \dots, d_ρ are *positive* integers. Note that here $\rho = \text{rank}(\mathbf{A})$.

Define $\mathbf{U} = \mathcal{I}_m$, $\mathbf{V} = \mathcal{I}_n$, and $\mathbf{D} = \mathbf{A}$. Then, \mathbf{U} and \mathbf{V} are unimodular, and we have $\mathbf{UAV} = \mathbf{D}$. We apply a sequence of elementary row and column operations to \mathbf{D} with the goal of reducing \mathbf{D} to a diagonal matrix of the form described above. For each row operation applied to \mathbf{D} , the same operation is also applied to \mathbf{U} ; and for each column operation applied to \mathbf{D} , the same operation is also applied to \mathbf{V} . Thus, the matrices \mathbf{U} and \mathbf{V} remain unimodular, and the relation $\mathbf{UAV} = \mathbf{D}$ remains unchanged (why?). When \mathbf{D} is reduced to the proper form, the process stops.

Let the elements of \mathbf{D} be denoted by d_{ij} . We locate an element d_{pq} that has the smallest nonzero absolute value among all elements of \mathbf{D} . This element is brought to the first row and the first column, by interchanging rows 1 and p , and columns 1 and q ; it becomes the new d_{11} . A suitable multiple of the first row is subtracted from row

i to reduce the element d_{i1} in absolute value as much as possible, $2 \leq i \leq m$. A suitable multiple of the first column is subtracted from column j to reduce the element d_{1j} in absolute value as much as possible, $2 \leq j \leq n$. A search is again made in the current state of \mathbf{D} for the element with the smallest nonzero absolute value. As before, it is brought to the first row and first column, and the row and column operations are repeated.

The process is continued until all elements of \mathbf{D} in the first row and all elements in the first column, except d_{11} , are zero. We then take the submatrix of \mathbf{D} obtained by deleting the first row and the first column, and process that matrix as we have processed \mathbf{D} . The algorithm stops when there is no submatrix to process, or when a submatrix obtained consists of zeros, whichever occurs first. At that point, the matrix \mathbf{D} has the desired form. During the entire process, every time a row operation is applied to \mathbf{D} , it is also applied to \mathbf{U} , and every time a column operation is applied to \mathbf{D} , it is also applied to \mathbf{V} .

The algorithm given below is a modification of the Smith normal form algorithm [Schr87]; it is also described in [D'Hol89].

Algorithm 2.3 (Diagonalization) Given an $m \times n$ integer matrix \mathbf{A} , this algorithm finds an $m \times m$ unimodular matrix \mathbf{U} , an $n \times n$ unimodular matrix \mathbf{V} , and an $m \times n$ diagonal matrix $\mathbf{D} = (d_{ij})$ of the form described above, such that $\mathbf{UAV} = \mathbf{D}$. We start with $\mathbf{U} = \mathcal{I}_m$, $\mathbf{V} = \mathcal{I}_n$, $\mathbf{D} = \mathbf{A}$, and work on these matrices. A variable k is defined such that the current submatrix of \mathbf{D} being processed is obtained from \mathbf{D} , by crossing out the topmost $k - 1$ rows and the leftmost $k - 1$ columns.

1. Set $\mathbf{U} \leftarrow \mathcal{I}_m$, $\mathbf{V} \leftarrow \mathcal{I}_n$, $\mathbf{D} \leftarrow \mathbf{A}$, $k \leftarrow 1$.
2. Among all elements d_{ij} with $k \leq i \leq m$ and $k \leq j \leq n$, locate an element d_{pq} that has the smallest nonzero absolute value. If no such element exists, then terminate the algorithm.
3. Interchange rows k and p in the matrix \mathbf{D} , and also in \mathbf{U} . Interchange columns k and q in \mathbf{D} , and also in \mathbf{V} . (The element d_{pq} of step 2 is now the element d_{kk} in the latest form of \mathbf{D} .)

- 4.** **do** $i = k + 1, m, 1$
- set**
- $\sigma \leftarrow \text{sig}(d_{ik} * d_{kk})$
- $z \leftarrow \lfloor |d_{ik}| / |d_{kk}| \rfloor$
- subtract σz (row k) from row i in **D**
- subtract σz (row k) from row i in **U**
- enddo**
- do** $j = k + 1, n, 1$
- set**
- $\sigma \leftarrow \text{sig}(d_{kj} * d_{kk})$
- $z \leftarrow \lfloor |d_{kj}| / |d_{kk}| \rfloor$
- subtract σz (col k) from col j in **D**
- subtract σz (col k) from col j in **V**
- enddo**
- 5.** If there is a nonzero element d_{ij} such that $i = k$ and $j > k$, or $i > k$ and $j = k$, then go to step 2.
- 6.** Increment k by 1. If $k \leq \min(m, n)$, then go to step 2; otherwise, terminate the algorithm. \square

Example 2.6 Let us diagonalize the 3×2 matrix

$$\mathbf{A} = \begin{pmatrix} 6 & 5 \\ 4 & 2 \\ 10 & -3 \end{pmatrix}.$$

The matrices **D**, **U** and **V** are initialized as follows:

$$\mathbf{D} \leftarrow \begin{pmatrix} 6 & 5 \\ 4 & 2 \\ 10 & -3 \end{pmatrix}, \quad \mathbf{U} \leftarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{V} \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

The variable k starts with the value 1; the algorithm ends when k becomes 3 which exceeds $\min(3, 2)$. The element in **D** with the smallest absolute value is $d_{22} = 2$ (i.e., $p = 2$ and $q = 2$). We start

by bringing this element to the first row and the first column.

Interchange rows 1 and 2 in \mathbf{D} and in \mathbf{U} :

$$\mathbf{D} \leftarrow \begin{pmatrix} 4 & 2 \\ 6 & 5 \\ 10 & -3 \end{pmatrix}, \quad \mathbf{U} \leftarrow \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Interchange columns 1 and 2 in \mathbf{D} and in \mathbf{V} :

$$\mathbf{D} \leftarrow \begin{pmatrix} 2 & 4 \\ 5 & 6 \\ -3 & 10 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{V} \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

In \mathbf{D} and in \mathbf{U} , subtract $[5/2] = 2$ times row 1 from row 2, and add $[3/2] = 1$ times row 1 to row 3:

$$\mathbf{D} \leftarrow \begin{pmatrix} 2 & 4 \\ 1 & -2 \\ -1 & 14 \end{pmatrix}, \quad \mathbf{U} \leftarrow \begin{pmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

In \mathbf{D} and in \mathbf{V} , subtract $[4/2] = 2$ times col 1 from col 2:

$$\mathbf{D} \leftarrow \begin{pmatrix} 2 & 0 \\ 1 & -4 \\ -1 & 16 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \quad \mathbf{V} \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix}.$$

There is no nonzero element besides d_{11} in row 1, but there are nonzero elements besides d_{11} in column 1. So, the process is repeated again with $k = 1$. Among the elements d_{ij} with $k \leq i \leq 3$ and $k \leq j \leq 2$, one element with the smallest nonzero absolute value is d_{21} . We choose this element, and then the new values of p and q are $p = 2$ and $q = 1$. We will bring d_{21} to the first row and the first column in \mathbf{D} .

Interchange rows 1 and 2 in \mathbf{D} and in \mathbf{U} :

$$\mathbf{D} \leftarrow \begin{pmatrix} 1 & -4 \\ 2 & 0 \\ -1 & 16 \end{pmatrix}, \quad \mathbf{U} \leftarrow \begin{pmatrix} 1 & -2 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix}.$$

In \mathbf{D} and in \mathbf{U} , subtract $[2/1] = 2$ times row 1 from row 2, and add $[1/1] = 1$ times row 1 to row 3:

$$\mathbf{D} \leftarrow \begin{pmatrix} 1 & -4 \\ 0 & 8 \\ 0 & 12 \end{pmatrix}, \quad \mathbf{U} \leftarrow \begin{pmatrix} 1 & -2 & 0 \\ -2 & 5 & 0 \\ 1 & -1 & 1 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix}.$$

In \mathbf{D} and in \mathbf{V} , add $[4/1] = 4$ times col 1 to col 2:

$$\mathbf{D} \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 8 \\ 0 & 12 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} 1 & -2 & 0 \\ -2 & 5 & 0 \\ 1 & -1 & 1 \end{pmatrix}, \quad \mathbf{V} \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}.$$

Since there are no nonzero elements d_{ij} such that $i = 1$ and $j > 1$, or $j = 1$ and $i > 1$, we increment k by 1 to 2. Among the elements d_{ij} with $k \leq i \leq 3$ and $k \leq j \leq 2$, the one with the smallest nonzero absolute value is already $d_{22} = 8$. Since now the elements to be processed are all in column 2, for the remainder of the algorithm the matrix \mathbf{V} remains unchanged. The matrices \mathbf{D} and \mathbf{U} are changed as follows:

Subtract 1 times row 2 from row 3 in \mathbf{D} and in \mathbf{U} :

$$\mathbf{D} \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 8 \\ 0 & 4 \end{pmatrix}, \quad \mathbf{U} \leftarrow \begin{pmatrix} 1 & -2 & 0 \\ -2 & 5 & 0 \\ 3 & -6 & 1 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}.$$

We need to bring the element ‘4’ to the position d_{22} . So, interchange rows 2 and 3 in \mathbf{D} and in \mathbf{U} :

$$\mathbf{D} \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 4 \\ 0 & 8 \end{pmatrix}, \quad \mathbf{U} \leftarrow \begin{pmatrix} 1 & -2 & 0 \\ 3 & -6 & 1 \\ -2 & 5 & 0 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}.$$

Subtract 2 times row 2 from row 3 in \mathbf{D} and in \mathbf{U} :

$$\mathbf{D} \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 4 \\ 0 & 0 \end{pmatrix}, \quad \mathbf{U} \leftarrow \begin{pmatrix} 1 & -2 & 0 \\ 3 & -6 & 1 \\ -8 & 17 & -2 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}.$$

There is no nonzero element besides d_{22} in row 2 or in column 2. The algorithm terminates here; check that $\mathbf{UAV} = \mathbf{D}$.

We already know that a product of a finite number of elementary matrices is a unimodular matrix. Now, we can prove the converse:

Lemma 2.3 *Each unimodular matrix can be expressed as a product of a finite number of elementary matrices.*

PROOF. Let \mathbf{A} denote a unimodular matrix. By Algorithm 2.3, we can find two sequences of elementary matrices $\mathbf{E}_1, \mathbf{E}_2, \dots, \mathbf{E}_M$ and $\mathbf{F}_1, \mathbf{F}_2, \dots, \mathbf{F}_N$, such that

$$\mathbf{D} = \mathbf{E}_M \cdots \mathbf{E}_2 \mathbf{E}_1 \mathbf{A} \mathbf{F}_1 \mathbf{F}_2 \cdots \mathbf{F}_N$$

is a diagonal matrix. Without any loss of generality, we may assume that the elements of \mathbf{D} on the main diagonal are nonnegative (why?). Note that the determinant of each matrix on the right hand side is ± 1 . Since the determinant of a product of matrices is the product of their determinants, we get $\det(\mathbf{D}) = \pm 1$. But $\det(\mathbf{D})$ is equal to the product of the elements on the main diagonal of \mathbf{D} . Hence, each such element must be 1; that is, \mathbf{D} is the identity matrix. Then, \mathbf{A} is the following product of elementary matrices:

$$\mathbf{A} = \mathbf{E}_1^{-1} \mathbf{E}_2^{-1} \cdots \mathbf{E}_M^{-1} \mathbf{F}_N^{-1} \cdots \mathbf{F}_2^{-1} \mathbf{F}_1^{-1}.$$

□

EXERCISES 2.4

1. Apply Algorithm 2.3 to the matrices of Exercise 2.3.3. Use additional steps, if necessary, to make the diagonal elements of \mathbf{D} all nonnegative.
2. In Example 2.6, find a set of matrices $\mathbf{U}_1, \mathbf{V}_1, \mathbf{D}_1$, such that \mathbf{U}_1 and \mathbf{V}_1 are unimodular, \mathbf{D}_1 is diagonal, $\mathbf{U}_1 \mathbf{A} \mathbf{V}_1 = \mathbf{D}_1$, and $(\mathbf{U}_1, \mathbf{V}_1, \mathbf{D}_1) \neq (\mathbf{U}, \mathbf{V}, \mathbf{D})$.
3. Modify Algorithm 2.3, so that for a given matrix \mathbf{A} , it finds two unimodular matrices \mathbf{U}, \mathbf{W} , and a diagonal matrix \mathbf{D} satisfying $\mathbf{U}\mathbf{A} = \mathbf{D}\mathbf{W}$. (Note that the idea is to find \mathbf{W} directly, as opposed to finding \mathbf{V} by Algorithm 2.3 and then computing $\mathbf{W} = \mathbf{V}^{-1}$.)
4. Learn about the Smith normal form of a matrix from [Schr87], and extend Algorithm 2.3 to compute this form.

5. Show that any unimodular matrix can be expressed as a product of reversal, interchange and *upper* skewing matrices only, and also as a product of reversal, interchange and *lower* skewing matrices only.
6. Show that the rank of an $m \times m$ unimodular matrix is m . Show also that for a given matrix \mathbf{A} , $\text{rank}(\mathbf{U}\mathbf{A}) = \text{rank}(\mathbf{A}\mathbf{V}) = \text{rank}(\mathbf{A})$ for any compatible unimodular matrices \mathbf{U} and \mathbf{V} .

2.5 Permutation Matrices

A *permutation* of a finite set is a one-to-one mapping of the set onto itself. A permutation matrix is any matrix that can be obtained by ‘permuting’ the columns (or rows) of an identity matrix. More formally, a *permutation matrix* is a square matrix such that

1. Each element is either 0 or 1;
2. Each row has exactly one 1;
3. Each column has exactly one 1.

Let \mathcal{P} denote any $m \times m$ permutation matrix. For $1 \leq i \leq m$, let $\pi(i)$ denote the number of the row containing the unique 1 in column i . Then, the function $\pi : i \mapsto \pi(i)$ is a permutation of the set $\{1, 2, \dots, m\}$, and it completely specifies the matrix \mathcal{P} . A compact notation³ for \mathcal{P} is

$$\mathcal{P} = \begin{bmatrix} 1 & 2 & \cdots & m \\ \pi(1) & \pi(2) & \cdots & \pi(m) \end{bmatrix}.$$

When \mathcal{P} acts on an m -vector \mathbf{x} , it permutes the components of \mathbf{x} in such a way that the i^{th} component of $\mathbf{x}\mathcal{P}$ is the $\pi(i)^{\text{th}}$ component of \mathbf{x} . For example, if \mathcal{P} denotes the permutation matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix} \equiv \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix},$$

then $(x_1, x_2, x_3)\mathcal{P} = (x_3, x_1, x_2)$, $(x_3, x_2, x_1)\mathcal{P} = (x_1, x_3, x_2)$, and $(6, 23, 12)\mathcal{P} = (12, 6, 23)$.

³This is not a standard notation. We use square brackets here so that a permutation matrix cannot be confused with an ordinary $2 \times m$ matrix.

An interchange matrix is a permutation matrix. A product of interchange matrices is clearly a permutation matrix. The converse is also true:

Lemma 2.4 *Every permutation matrix can be expressed as a product of interchange matrices.*

For a proof of this lemma, see any standard algebra text. We can even go one step further:

Corollary 1 *Every permutation matrix can be expressed as a product of interchange matrices each of which interchanges two adjacent columns (or two adjacent rows).*

PROOF. An interchange matrix is obtained from the identity matrix by interchanging two columns. An interchange of columns can always be accomplished by a sequence of adjacent interchanges. For example, to interchange columns r and s , with $r < s$, we can use the following sequence of interchanges involving adjacent column pairs:

$$(r, r+1), (r+1, r+2), \dots, (s-1, s), (s-2, s-1), \dots, (r, r+1).$$

Thus, an interchange matrix can be expressed as a product of special interchange matrices that interchange two adjacent columns (Lemma 2.1). Since a permutation matrix is a product of general interchange matrices, the result follows. \square

Corollary 2 *Every permutation matrix is a unimodular matrix.*

PROOF. Since an interchange matrix is unimodular, and a product of unimodular matrices is unimodular, a permutation matrix is necessarily unimodular. \square

There are $m!$ distinct $m \times m$ permutation matrices corresponding to the $m!$ distinct permutations of the set $\{1, 2, \dots, m\}$. The identity matrix corresponds to the *trivial* permutation, that is, the identity function. The six 3×3 permutation matrices are: the 3×3 identity matrix, the three interchange matrices:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix}, \quad \begin{bmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{bmatrix}, \quad \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix};$$

and the matrices:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}.$$

Note that the first two interchange matrices interchange adjacent columns (or rows). We can express every 3×3 permutation matrix as a product of these two. For example,

$$\begin{aligned} \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix} &\equiv \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &\equiv \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix}. \end{aligned}$$

EXERCISES 2.5

- Express each 3×3 permutation matrix as a product of the two interchange matrices that interchange adjacent columns.
- Write down all the 4×4 permutation matrices, using both the standard and the compact notations. Express each permutation matrix as a product of the special interchange matrices that interchange adjacent columns. Are those expressions unique?
- If \mathcal{P} is an $m \times m$ permutation matrix and σ a direction vector of size m , then $\sigma\mathcal{P}$ is also a direction vector of size m . For each permutation matrix \mathcal{P} given below, find all positive direction vectors σ (of the right size) such that the direction vector $\sigma\mathcal{P}$ is negative:
 - The 2×2 permutation matrices
 - The 3×3 permutation matrices
 - $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 3 & 1 \end{bmatrix}$
 - $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{bmatrix}$.

Use direction vector forms (Section 1.3).

Chapter 3

Linear Equations and Inequalities

3.1 Introduction

Equations and inequalities in integer variables arise naturally in the dependence analysis of a program, when we try to analyze the memory reference pattern. The equations are created by matching the corresponding subscripts of instances of two subscripted variables that are elements of the same array. The variables in these equations are integer variables, since they come from the index variables of loops, which take integer values. A polynomial equation in integer variables is called a *diophantine equation*.¹ The array subscripts are usually linear (affine) functions of the index variables, with integer coefficients; consequently, they lead to *linear diophantine equations* with *integer* coefficients.

The loop limits bounding the index variables of loops generate a system of inequalities usually involving the same variables that appear in the diophantine equations mentioned above. In most real programs, a limit of a loop L is either a constant or a very simple linear (affine) function of the index variables of loops containing L . Hence, the corresponding inequalities are also linear, and have integer coefficients. In a typical dependence problem, there is an ad-

¹Named after Diophantos of Alexandria (third century).

ditional set of equations and inequalities, that represents the strict ordering of two references to the same memory location.

For an example, consider the double loop:

```

do  $I_1 = 1, 100$ 
  do  $I_2 = 1, 200$ 
     $S : \quad X(2I_1 + 2I_2 - 3) = Y(I_1, I_2 + 1) - 5$ 
     $T : \quad Z(I_1 + 1, I_2) = X(6I_1 + 4I_2 - 2) - 2$ 
  enddo
enddo

```

For a value (i_1, i_2) of (I_1, I_2) , the corresponding instance of statement S is denoted by $S(i_1, i_2)$, and for a value (j_1, j_2) of (I_1, I_2) , the corresponding instance of statement T is denoted by $T(j_1, j_2)$. The definition of dependence (Chapter 4) says that if T depends on S , then there is an instance $S(i_1, i_2)$ of S and an instance $T(j_1, j_2)$ of T , such that both instances reference the same memory location, and the reference by $S(i_1, i_2)$ precedes the reference by $T(j_1, j_2)$ in the (sequential) execution of the program.

Note that $X(2i_1 + 2i_2 - 3)$ is the output variable of the instance $S(i_1, i_2)$, and $X(6j_1 + 4j_2 - 2)$ is the input variable of the instance $T(j_1, j_2)$. These two variables will represent the same memory location iff

$$2i_1 + 2i_2 - 3 = 6j_1 + 4j_2 - 2.$$

Since i_1, j_1 are values of the index variable I_1 , they must lie between 1 and 100. Since i_2, j_2 are values of the index variable I_2 , they must lie between 1 and 200. The instance $S(i_1, i_2)$ will write this memory location *before* the instance $T(j_1, j_2)$ reads it, iff $(i_1, i_2) \preceq (j_1, j_2)$. (See Example 1.3. The case $(i_1, i_2) = (j_1, j_2)$ is included here since S appears lexically before T in the program.) Thus, if T depends on S , then there are integers i_1, j_1, i_2, j_2 such that

$$2i_1 + 2i_2 - 3 = 6j_1 + 4j_2 - 2 \tag{3.1}$$

$$\left. \begin{array}{l} 1 \leq i_1 \leq 100, \quad 1 \leq j_1 \leq 100 \\ 1 \leq i_2 \leq 200, \quad 1 \leq j_2 \leq 200 \end{array} \right\}$$

and

$$i_1 < j_1 \text{ or } i_1 = j_1, i_2 < j_2 \text{ or } i_1 = j_1, i_2 = j_2.$$

First, consider the equation without the extra constraints. It is easy to see that (3.1) has infinitely many *real* solutions. For instance, we may take any set of values for j_1, i_1, i_2 and then choose

$$i_1 = (6j_1 - 2i_2 + 4j_2 + 1)/2.$$

To decide if there are any *integer* solutions, note that the greatest common divisor of the coefficients on the left hand side of (3.1) is 2. If an integer solution exists, then 2 must (evenly) divide the right hand side, which is impossible. Hence, there are no integer solutions to (3.1), so that T does not depend on S in the given program. (The constraints on the solution of (3.1) do not matter in this particular case.)

A linear diophantine equation of the form (3.1) has an integer solution iff the greatest common divisor of the coefficients on the left hand side divides the right hand side. In general, the number of equations we get is equal to the number of dimensions of the array involved. For a system of two or more equations, the condition for the existence of an integer solution is more complicated. To decide if a general system of linear diophantine equations has an integer solution, we need to use the echelon reduction or the diagonalization algorithm of Chapter 2. These algorithms also generate all solutions when solutions exist.

If the system of equations is found to have a solution, the next step is to decide if there is a solution that also satisfies the other constraints of the problem. A system of linear inequalities in integer variables is generally much harder to solve than a system of linear diophantine equations. A discussion of various integer programming techniques for solving inequalities is certainly beyond our scope. In this chapter, we will only describe a straightforward method called the Fourier elimination algorithm. Although it is not a practical algorithm for a large system, the Fourier method is quite adequate for most loop transformation problems, where the inequalities are usually simple and few in number. The major objection to this method is that it decides the existence of a *real* solution to a system of linear inequalities. When a real solution exists, the question of the existence of an integer solution is still left open. The Fourier

method is also useful in loop transformation problems other than those related to dependence analysis. Sometimes, we may already *know* that there are integer solutions to a system of inequalities, and merely want to express the solution-set in a certain form. A typical example is finding the new loop limits after a nest of loops has been transformed by a certain loop transformation; we faced this problem in Example 2.1. In such a situation, the elimination method provides a convenient set of formulas describing the solution-set.

In Section 3.2, we introduce notations for the positive and negative parts of a number. Using these notations, one can easily write compact expressions that contain many cases based on various combinations of possible signs of different variables. General linear diophantine equations are discussed in Section 3.4. To solve these equations, we need to use unimodular matrices and greatest common divisors. Unimodular matrices were covered in detail in Chapter 2; the basic properties of the greatest common divisors are given in Section 3.3. Because of its importance, a single linear diophantine equation in two variables has been given a detailed treatment in Section 3.5. The last section deals with the Fourier method of elimination.

We recommend [Schr87] as the general reference for this chapter. This book also has a very comprehensive bibliography that can be used for further studies in diophantine equations and the elimination method. In particular, one should read [Kert81] for diophantine equations, and [Duff74] and [Will86] for the Fourier method.

3.2 Parts of a Number

The *positive part* a^+ and the *negative part* a^- of a real number a are defined by

$$\begin{aligned} a^+ &= \max(a, 0) \\ a^- &= \max(-a, 0). \end{aligned}$$

In other words, $a^+ = a$ and $a^- = 0$ for $a \geq 0$, while $a^+ = 0$ and $a^- = -a$ for $a \leq 0$. Thus, $4^+ = 4$, $4^- = 0$, $(-4)^+ = 0$, and

$(-4)^- = 4$. The following lemma is from [Bane76]; it lists the basic properties of positive and negative parts.

Lemma 3.1 *For any real number a , the following statements hold:*

- (a) $a^+ \geq 0, \quad a^- \geq 0$
- (b) $a = a^+ - a^-, \quad |a| = a^+ + a^-$
- (c) $(-a)^+ = a^-, \quad (-a)^- = a^+$
- (d) $(a^+)^+ = a^+, \quad (a^+)^- = 0$
- (e) $(a^-)^+ = a^-, \quad (a^-)^- = 0$
- (f) $-a^- \leq a \leq a^+$.

The next lemma gives convenient expressions for the extreme values of a simple function. Many useful formulas of this type can be derived from these results; see [Bane76] and [Bane88a].

Lemma 3.2 *The minimum and maximum values of a function of the form $f(x) = ax$ in a closed interval $[p, q]$ are $(a^+p - a^-q)$ and $(a^+q - a^-p)$, respectively.*

PROOF. For $p \leq x \leq q$, we have

$$\begin{array}{lcl} a^+p & \leq & a^+x \\ -a^-q & \leq & -a^-x \end{array} \quad \begin{array}{lcl} a^+x & \leq & a^+q \\ -a^-x & \leq & -a^-p \end{array}$$

since $a^+ \geq 0$ and $-a^- \leq 0$. Adding these two sets of inequalities, we get

$$a^+p - a^-q \leq ax \leq a^+q - a^-p$$

since $a = a^+ - a^-$.

To complete the proof, note that these bounds for $f(x)$ are actually attained at the end points $x = p$ and $x = q$. For example, when $a \geq 0$, we have

$$\begin{aligned} a^+p - a^-q &= ap &= f(p) \\ a^+q - a^-p &= aq &= f(q). \end{aligned}$$

□

Example 3.1 The function $g(x) = 2x$ is monotonically increasing; its minimum value in the interval $-1 \leq x \leq 2$ is attained at $x = -1$ and the maximum value at $x = 2$. The function $h(x) = -2x$ is monotonically decreasing; its minimum value in the same interval is attained at $x = 2$, and the maximum value at $x = -1$. Both cases are handled by Lemma 3.2: it gives the minimum and maximum values of the function $f(x) = ax$ in $[-1, 2]$ to be $(-a^+ - 2a^-)$ and $(2a^+ + a^-)$, respectively.

EXERCISES 3.2

1. Prove Lemma 3.1.
2. Let a and b denote arbitrary real numbers. Find the extreme values of the function $f(x, y) = ax + by$ in a region \mathcal{R} defined by a pair of inequalities of the form:
 - (a) $p_1 \leq x \leq q_1, p_2 \leq y \leq q_2$
 - (b) $p \leq x \leq q, p \leq y \leq x$
 - (c) $p \leq x \leq q, x \leq y \leq q$.
3. Using the results of the previous exercise, find the extreme values of $f(x, y) = -2x + 3y$ in the following regions:
 - (a) $-4 \leq x \leq 2, 3 \leq y \leq 17$
 - (b) $0 \leq x \leq 5, 0 \leq y \leq x$
 - (c) $-15 \leq x \leq 10, x \leq y \leq 10$.
4. Let \mathbf{R} denote the real line and m a positive integer. Define a function $f : \mathbf{R}^m \rightarrow \mathbf{R}$ by $f(x_1, x_2, \dots, x_m) = \sum_{k=1}^m a_k x_k$. Let c denote a real number, and \mathcal{R} an m -dimensional ‘rectangle’:

$$\{(x_1, x_2, \dots, x_m) : p_1 \leq x_1 \leq q_1, p_2 \leq x_2 \leq q_2, \dots, p_m \leq x_m \leq q_m\}.$$

Prove that there is a real vector $\mathbf{x} = (x_1, x_2, \dots, x_m)$ in \mathcal{R} such that $f(\mathbf{x}) = c$ iff

$$\sum_{k=1}^m (a_k^+ p_k - a_k^- q_k) \leq c \leq \sum_{k=1}^m (a_k^+ q_k - a_k^- p_k).$$

Suppose now that c and all other constants are integers. If the above condition holds, will there exist an *integer* vector \mathbf{x} in \mathcal{R} satisfying $f(\mathbf{x}) = c$? Either prove, or disprove by giving a counterexample.

3.3 Greatest Common Divisor

If a and b denote two integers such that b is not zero, then the notation a/b will represent the rational number $\frac{a}{b}$. We say that b (*evenly*) *divides* a if a/b is an integer (i.e., $a \bmod b = 0$).

For a list of integers a_1, a_2, \dots, a_m , not all zero, their *greatest common divisor*, $\gcd(a_1, a_2, \dots, a_m)$, is the largest positive integer that divides all numbers in the list. The gcd of a list of zeros is defined to be 0. The integers a_1, a_2, \dots, a_m are *relatively prime* if their gcd is 1. The elementary properties of the gcd are stated below:

Lemma 3.3 *For a list of integers a_1, a_2, \dots, a_m , the following statements hold:*

- (a) $\gcd(a_1, a_2, \dots, a_m) = \gcd(\pm a_1, \pm a_2, \dots, \pm a_m)$;
- (b) $\gcd(0, a_1, a_2, \dots, a_m) = \gcd(a_1, a_2, \dots, a_m)$;
- (c) $\gcd(1, a_1, a_2, \dots, a_m) = 1$;
- (d) $\gcd(a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(m)}) = \gcd(a_1, a_2, \dots, a_m)$
where π is any permutation of the set $\{1, 2, \dots, m\}$;
- (e) $\gcd(pa_1, pa_2, \dots, pa_m) = |p| \cdot \gcd(a_1, a_2, \dots, a_m)$
for any integer p ;
- (f) $\gcd(a_1 - qa_2, a_2, \dots, a_m) = \gcd(a_1, a_2, \dots, a_m)$
for any integer q ;
- (g) If a_1, a_2, \dots, a_m are not all zero, then $a_1/g, a_2/g, \dots, a_m/g$ are relatively prime, where $g = \gcd(a_1, a_2, \dots, a_m)$.

The gcd of a list of integers can be found by a direct application of either Echelon Reduction algorithm (Algorithm 2.1 or 2.2):

Theorem 3.4 *Let a_1, a_2, \dots, a_m denote a given list of integers. Let \mathbf{U} be an $m \times m$ unimodular matrix and $\mathbf{S} = (s_{11}, 0, \dots, 0)'$ an $m \times 1$ echelon matrix, such that $\mathbf{U}\mathbf{A} = \mathbf{S}$, where \mathbf{A} is the $m \times 1$ matrix $(a_1, a_2, \dots, a_m)'$. Then, $\gcd(a_1, a_2, \dots, a_m) = |s_{11}|$.*

PROOF. Let $g = \gcd(a_1, a_2, \dots, a_m)$ and let $\mathbf{U} = (u_{ij})$. The first scalar equation in the system represented by the matrix equation $\mathbf{U}\mathbf{A} = \mathbf{S}$ is

$$u_{11}a_1 + u_{12}a_2 + \cdots + u_{1m}a_m = s_{11}. \quad (3.2)$$

If a_1, a_2, \dots, a_m are all equal to zero, then s_{11} must be zero, and there is nothing left to prove. Assume, therefore, that the a_i 's are not all zero, so that $g > 0$. Since g divides each a_i , it follows from (3.2) that g must also divide s_{11} .

The inverse \mathbf{U}^{-1} of \mathbf{U} is also an integer matrix. Letting $\mathbf{U}^{-1} = (v_{ij})$, we get from the equation $\mathbf{U}^{-1}\mathbf{S} = \mathbf{A}$ that

$$v_{i1}s_{11} = a_i \quad \text{for } 1 \leq i \leq m.$$

Since the a_i 's are not all zero, s_{11} cannot be zero. As s_{11} divides each a_i , it must also divide $g = \gcd(a_1, a_2, \dots, a_m)$.

Since s_{11} and g are integers such that each divides the other, and g is positive, it follows that $g = |s_{11}|$. \square

Corollary 1 *Let \mathbf{V} be an $m \times m$ unimodular matrix and $\mathbf{S} = (s_{11}, 0, \dots, 0)'$ an $m \times 1$ echelon matrix, such that $\mathbf{A} = \mathbf{VS}$, where \mathbf{A} is the $m \times 1$ matrix $(a_1, a_2, \dots, a_m)'$. Then, $\gcd(a_1, a_2, \dots, a_m) = |s_{11}|$.*

Corollary 2 *If g denotes the gcd of a list of integers a_1, a_2, \dots, a_m , then there exist integers t_1, t_2, \dots, t_m such that*

$$g = t_1a_1 + t_2a_2 + \cdots + t_ma_m.$$

PROOF. Multiplying (3.2) by $\text{sig}(s_{11})$, we get

$$g = |s_{11}| = \text{sig}(s_{11})s_{11} = \sum_{j=1}^m t_j a_j$$

where $t_j = \text{sig}(s_{11})u_{1j}$, $1 \leq j \leq m$. \square

As the following example shows, such an expression for the gcd is not unique.

Example 3.2 To find $\gcd(4, 6, 4)$, we apply Algorithm 2.1 to the 3×1 matrix $\mathbf{A} = (4, 6, 4)'$, and get the matrices

$$\mathbf{U} = \begin{pmatrix} 0 & 1 & -1 \\ 1 & -2 & 2 \\ 0 & -2 & 3 \end{pmatrix} \text{ and } \mathbf{S} = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix},$$

such that \mathbf{U} is unimodular, \mathbf{S} is echelon, and $\mathbf{U}\mathbf{A} = \mathbf{S}$. This gives $\gcd(4, 6, 4) = |\det(\mathbf{U})| = 2$ (Theorem 3.4), and the expression (from (3.2)):

$$2 = (0)4 + (1)6 + (-1)4.$$

Note that the unimodular matrix

$$\mathbf{U}_1 = \begin{pmatrix} 1 & -1 & 1 \\ 1 & -2 & 2 \\ 0 & -2 & 3 \end{pmatrix}$$

and the same echelon matrix \mathbf{S} also satisfy the relation $\mathbf{U}_1\mathbf{A} = \mathbf{S}$. Using \mathbf{U}_1 , we get a different expression for the gcd from (3.2):

$$2 = (1)4 + (-1)6 + (1)4.$$

(Does the reader see how we got the matrix \mathbf{U}_1 ?)

Corollary 3 *The integers forming a row or a column of a unimodular matrix are relatively prime.*

PROOF. Let \mathbf{U} denote any unimodular matrix, so that $\det(\mathbf{U}) = \pm 1$. We can express $\det(\mathbf{U})$ as a linear combination of the elements of any row or column of \mathbf{U} . Then, the gcd of those elements must divide $|\det(\mathbf{U})|$, and hence must be equal to 1. \square

Corollary 4 *For any nonzero integer vector (a_1, a_2, \dots, a_m) with $\gcd(a_1, a_2, \dots, a_m) = g$, there exists an $m \times m$ unimodular matrix \mathbf{V} whose first column is the vector $(a_1/g, a_2/g, \dots, a_m/g)$.*

PROOF. Let $\mathbf{A} = (a_1, a_2, \dots, a_m)'$. By Algorithm 2.2, we can find a unimodular matrix $\mathbf{V} = (v_{ij})$ and an echelon matrix $\mathbf{S} = (s_{11}, 0, \dots, 0)'$, such that $\mathbf{A} = \mathbf{VS}$. We can always choose \mathbf{V} and \mathbf{S} such that $s_{11} > 0$. Assuming that it is so, we have $s_{11} = g$ and $\mathbf{S} = (g, 0, \dots, 0)'$ (Corollary 1). The relation $\mathbf{A} = \mathbf{VS}$ is then equivalent to

$$(a_1, a_2, \dots, a_m)' = (v_{11}g, v_{21}g, \dots, v_{m1}g)'.$$

Thus, $(a_1/g, a_2/g, \dots, a_m/g)$ is the first column of the unimodular matrix \mathbf{V} . \square

There is nothing special about the ‘first column’ in the above corollary. For any j in $1 \leq j \leq m$, we can interchange the columns 1 and j in \mathbf{V} to get a unimodular matrix whose j^{th} column is $(a_1/g, a_2/g, \dots, a_m/g)$. Also, since the transpose of a unimodular matrix is unimodular, we can easily find a unimodular matrix \mathbf{V} such that any specified *row* of \mathbf{V} is $(a_1/g, a_2/g, \dots, a_m/g)$.

The classical Euclid’s algorithm [Knut73] finds the gcd of two integers. We can find the gcd of a list of m integers by $m - 1$ applications of that algorithm, repeatedly using the property:

$$\gcd(a_1, a_2, \dots, a_m) = \gcd(a_1, \gcd(a_2, \dots, a_m))$$

for any $m \geq 3$.

The Extended Euclid’s algorithm [Knut73] finds the gcd of two integers a, b , and an expression for the gcd as a linear combination of a and b . It can be easily derived as a special case of Algorithm 2.1 for the 2×1 matrix $\mathbf{A} = (a, b)'$; we omit the details.

EXERCISES 3.3

- Find the gcd of each list of integers, and an expression for the gcd as a linear combination of those integers:
 - 34, 55
 - 89, 233
 - 34, 55, 89
 - 24, 60, 100, -170

- (e) 144, 233, 377, 610
(f) -57, 57, 38, -76.
2. For each list in the previous exercise, find a unimodular matrix whose second row is equal to the integer vector formed by that list, divided by the corresponding gcd.
 3. Show that for an $m \times m$ unimodular matrix, the gcd of all $k \times k$ subdeterminants is 1, for any k in $1 \leq k \leq m$.
 4. Show that for a given matrix, the gcd of all subdeterminants of a fixed size is invariant under any elementary row or column operation on the matrix.

3.4 Linear Diophantine Equations

When we talk about diophantine equations, a *solution* is always an *integer* solution, unless otherwise indicated. We are interested in solving a system of linear diophantine equations with integer coefficients. First, we study the special case of a single equation, and then extend the results to a general system.

Consider a linear diophantine equation

$$a_1x_1 + a_2x_2 + \cdots + a_mx_m = c$$

in m variables, with integer coefficients: a_1, a_2, \dots, a_m , not all zero, and an integer right-hand side c . Note that the case $m = 1$ is trivial to solve: it is easy to decide if there is an integer x_1 such that $a_1x_1 = c$. We will solve the general equation by reducing the problem to this trivial case. Let g denote the gcd of a_1, a_2, \dots, a_m .

The matrix form of the above equation is

$$\mathbf{x}\mathbf{A} = c \tag{3.3}$$

where $\mathbf{x} = (x_1, x_2, \dots, x_m)$ and \mathbf{A} denotes the *coefficient matrix* $(a_1, a_2, \dots, a_m)'$. By Algorithm 2.1, we can find a unimodular matrix \mathbf{U} and an echelon matrix $\mathbf{S} = (s_{ij})$, such that $\mathbf{U}\mathbf{A} = \mathbf{S}$. Assume that the matrices have been so chosen that s_{11} is positive. Then, s_{11} gives g , and we have $\mathbf{S} = (g, 0, \dots, 0)'$ (Theorem 3.4). The following theorem will show that equation (3.3) has a solution iff the

trivial diophantine equation $gt_1 = c$ has a solution. Moreover, when a solution exists, there is a formula (the *general solution*) involving \mathbf{U} that gives *all* the solutions.

As we know, the choice of the matrix \mathbf{U} is not unique (Example 3.2). The *form* of the general solution depends on the particular \mathbf{U} used, but the *set* of all solutions is independent of the choice.

Theorem 3.5 *The linear diophantine equation (3.3) has a solution iff the gcd g of its coefficients divides c . When a solution exists, the set of all solutions is given by the formula (the general solution):*

$$\mathbf{x} = (c/g, t_2, t_3, \dots, t_m) \cdot \mathbf{U} \quad (3.4)$$

where t_2, t_3, \dots, t_m are arbitrary integers, and \mathbf{U} is any $m \times m$ unimodular matrix such that $\mathbf{U}\mathbf{A} = (g, 0, \dots, 0)'$.

PROOF. Let \mathbf{U} denote an $m \times m$ unimodular matrix such $\mathbf{U}\mathbf{A} = (g, 0, \dots, 0)'$. Any m -vector $\mathbf{x} = (x_1, x_2, \dots, x_m)$ can be written in the form $\mathbf{x} = \mathbf{t}\mathbf{U}$, where \mathbf{t} is the m -vector $\mathbf{x} \cdot \mathbf{U}^{-1}$. Since \mathbf{U} and \mathbf{U}^{-1} are integral, \mathbf{x} is integral iff \mathbf{t} is integral.

A vector $\mathbf{x} = (t_1, t_2, \dots, t_m) \cdot \mathbf{U}$ will satisfy (3.3) iff

$$\begin{aligned} c &= \mathbf{x}\mathbf{A} \\ &= (t_1, t_2, \dots, t_m) \cdot \mathbf{U}\mathbf{A} \\ &= (t_1, t_2, \dots, t_m) \cdot (g, 0, \dots, 0)' \\ &= gt_1. \end{aligned}$$

If g does not divide c , then c/g is not an integer, and there is no (integer) solution to (3.3). If g divides c , then any integer vector of the form $\mathbf{x} = (t_1, t_2, \dots, t_m) \cdot \mathbf{U}$ is a solution, provided $t_1 = c/g$. \square

Note that equation (3.3) has a solution if one of the coefficients is ± 1 , since then $g = 1$ (Lemma 3.3(c)). For example, if $a_1 = 1$, then the general solution is

$$(x_1, x_2, \dots, x_m) = (c - a_2x_2 - \cdots - a_mx_m, x_2, \dots, x_m)$$

where x_2, \dots, x_m are arbitrary integers.

The following corollary is a version of the two-variable case of the above theorem; it is a well-known result of elementary number theory.

Corollary 1 *Let a, b, c denote integers such that a and b are not both zero, and let $g = \gcd(a, b)$. The linear diophantine equation*

$$ax + by = c$$

has a solution iff g divides c . When a solution exists, the general solution is given by

$$\begin{aligned} x &= (b/g)t + (c/g)x_0 \\ y &= -(a/g)t + (c/g)y_0 \end{aligned}$$

where t is an arbitrary integer, and (x_0, y_0) is any pair of integers such that $g = ax_0 + by_0$.

PROOF. We need only prove the formula for the general solution when solutions exist. Define

$$\mathbf{U} = \begin{pmatrix} x_0 & y_0 \\ b/g & -a/g \end{pmatrix}.$$

Then, \mathbf{U} is unimodular and $\mathbf{U} \cdot (a, b)' = (g, 0)'$. By Theorem 3.5, the general solution to the given equation is $(x, y) = (c/g, t) \cdot \mathbf{U}$ which gives the formula in the corollary. \square

Example 3.3 We will apply Theorem 3.5 to solve the diophantine equation:

$$4x_1 + 6x_2 + 4x_3 = 8.$$

To decide if a solution exists, we need to know $g = \gcd(4, 6, 4)$. In case it does, to get the general solution, we need to find a 3×3 unimodular matrix \mathbf{U} such that

$$\mathbf{U} \cdot \begin{pmatrix} 4 \\ 6 \\ 4 \end{pmatrix} = \begin{pmatrix} g \\ 0 \\ 0 \end{pmatrix}.$$

Both the gcd and the matrix \mathbf{U} are computed by Algorithm 2.1. As in Example 3.2, we find two matrices

$$\mathbf{U} = \begin{pmatrix} 0 & 1 & -1 \\ 1 & -2 & 2 \\ 0 & -2 & 3 \end{pmatrix} \quad \text{and} \quad \mathbf{S} = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}$$

such that \mathbf{U} is unimodular, \mathbf{S} is echelon, and $\mathbf{U}\mathbf{A} = \mathbf{S}$. Since the element in the first row of \mathbf{S} is positive, it gives g (Theorem 3.4). Because 2 divides the right-hand side 8, the equation of this example has a solution. The general solution is given in terms of two integer parameters:

$$\begin{aligned} (x_1, x_2, x_3) &= (8/2, t_2, t_3) \cdot \mathbf{U} \\ &= (t_2, 4 - 2t_2 - 2t_3, -4 + 2t_2 + 3t_3). \end{aligned}$$

Note that if we use the matrix \mathbf{U}_1 of Example 3.2, then we would get a ‘different’ general solution. It would, however, generate the same solution-set as that represented by the general solution given above (proof?).

Next, we consider a system of n linear diophantine equations in m variables:

$$\begin{aligned} a_{11}x_1 + a_{21}x_2 + \cdots + a_{m1}x_m &= c_1 \\ a_{12}x_1 + a_{22}x_2 + \cdots + a_{m2}x_m &= c_2 \\ &\vdots \\ a_{1n}x_1 + a_{2n}x_2 + \cdots + a_{mn}x_m &= c_n \end{aligned}$$

where the coefficients and the right-hand sides are all integers. In matrix notation, this system may be written as

$$\mathbf{x}\mathbf{A} = \mathbf{c}$$

where $\mathbf{x} = (x_1, x_2, \dots, x_m)$, $\mathbf{c} = (c_1, c_2, \dots, c_n)$, and \mathbf{A} denotes the $m \times n$ coefficient matrix (a_{ij}) .

Such a system is easy to solve if the matrix of coefficients is in echelon or diagonal form. For example, to see that the system

$$(x_1, x_2) \cdot \begin{pmatrix} 3 & 2 \\ 0 & 5 \end{pmatrix} = (6, 7)$$

or

$$\left. \begin{array}{rcl} 3x_1 & = & 6 \\ 2x_1 + 5x_2 & = & 7 \end{array} \right\}$$

has no solution, we solve the first equation and get $x_1 = 2$. The second equation then becomes $5x_2 = 3$ which clearly has no solution in integers. Given a system of equations $\mathbf{x}\mathbf{A} = \mathbf{c}$, we can reduce the coefficient matrix \mathbf{A} to echelon or diagonal form by the echelon reduction or the diagonalization algorithm, respectively. The following two theorems show that the solution of the original system is directly related to the solution of the transformed system (which is simpler) in each case.

Theorem 3.6 *Let \mathbf{A} be a given $m \times n$ integer matrix and \mathbf{c} a given integer n -vector. Let \mathbf{U} denote an $m \times m$ unimodular matrix and \mathbf{S} an $m \times n$ echelon matrix, such that $\mathbf{U}\mathbf{A} = \mathbf{S}$. The system of equations*

$$\mathbf{x}\mathbf{A} = \mathbf{c} \tag{3.5}$$

has a solution iff there exists an integer m -vector \mathbf{t} such that $\mathbf{t}\mathbf{S} = \mathbf{c}$. When a solution exists, the set of all solutions is given by the formula (the general solution)

$$\mathbf{x} = \mathbf{t}\mathbf{U}$$

where \mathbf{t} is any integer vector satisfying $\mathbf{t}\mathbf{S} = \mathbf{c}$.

PROOF. As in the proof of Theorem 3.5, an m -vector $\mathbf{x} = \mathbf{t}\mathbf{U}$ will be a solution to equation (3.5) iff

$$\mathbf{c} = \mathbf{x}\mathbf{A} = \mathbf{t}\mathbf{U}\mathbf{A} = \mathbf{t}\mathbf{S}.$$

If there is no integer vector \mathbf{t} such that $\mathbf{t}\mathbf{S} = \mathbf{c}$, then there is no (integer) solution to (3.5). If there is such a \mathbf{t} , then all solutions have the form $\mathbf{x} = \mathbf{t}\mathbf{U}$, where \mathbf{t} is integral and $\mathbf{t}\mathbf{S} = \mathbf{c}$. \square

The proof of the following theorem is similar (Exercise 3).

Theorem 3.7 Let \mathbf{A} be a given $m \times n$ integer matrix and \mathbf{c} a given integer n -vector. Let \mathbf{U} denote an $m \times m$ unimodular matrix, \mathbf{V} an $n \times n$ unimodular matrix, and \mathbf{D} an $m \times n$ diagonal matrix, such that $\mathbf{UAV} = \mathbf{D}$.

The system (3.5) has a solution iff there is an integer m -vector \mathbf{t} such that $\mathbf{tD} = \mathbf{cV}$. When a solution exists, all solutions are given by the formula (the general solution)

$$\mathbf{x} = \mathbf{tU}$$

where \mathbf{t} is any integer vector satisfying $\mathbf{tD} = \mathbf{cV}$.

For a given system of diophantine equations $\mathbf{xA} = \mathbf{c}$ in m variables, the solution-set is a certain fixed subset of \mathbf{Z}^m . If there are no solutions, then that set is, of course, empty. Assume that solutions do exist. Forms of general solutions produced by different solution methods may be different, but all general solutions must generate the same solution-set. Let ρ denote the rank of the coefficient matrix \mathbf{A} . Note that exactly ρ components of the integer vector \mathbf{t} are determined by the equation $\mathbf{tS} = \mathbf{c}$ in Theorem 3.6, or the equation $\mathbf{tD} = \mathbf{cV}$ in Theorem 3.7 (proof?). Thus, the number of components that are undetermined (i.e., the number of independent integer parameters) in any general solution is equal to $m - \rho$. We get a unique solution (one without any parameters) if $\rho = m$. The total number of solutions to a system of linear diophantine equations is either zero, one, or infinite.

Example 3.4 We will solve the system of diophantine equations

$$\left. \begin{array}{rcl} 2x_1 + 6x_2 + 4x_3 - 2x_4 & = & 4 \\ x_1 + 3x_2 & & + 5x_4 & = & 2 \\ -2x_2 + 3x_3 - x_4 & = & 8 \end{array} \right\} \quad (3.6)$$

or $\mathbf{xA} = \mathbf{c}$ where $\mathbf{x} = (x_1, x_2, x_3, x_4)$, $\mathbf{c} = (4, 2, 8)$ and

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & 0 \\ 6 & 3 & -2 \\ 4 & 0 & 3 \\ -2 & 5 & -1 \end{pmatrix}.$$

By Algorithm 2.1, we find two matrices

$$\mathbf{U} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & -1 & 2 & 1 \\ -3 & 1 & 0 & 0 \\ 17 & -4 & -3 & -1 \end{pmatrix} \text{ and } \mathbf{S} = \begin{pmatrix} -2 & 5 & -1 \\ 0 & 2 & 7 \\ 0 & 0 & -2 \\ 0 & 0 & 0 \end{pmatrix}$$

such that \mathbf{U} is unimodular, \mathbf{S} is echelon, and $\mathbf{U}\mathbf{A} = \mathbf{S}$. Solving the reduced system

$$(t_1, t_2, t_3, t_4) \cdot \mathbf{S} = (4, 2, 8)$$

we get $t_1 = -2, t_2 = 6, t_3 = 18$; and t_4 remains undetermined. By Theorem 3.6, the general solution to system (3.6) is

$$\mathbf{x} = \mathbf{t}\mathbf{U} = (-54 + 17t_4, 12 - 4t_4, 12 - 3t_4, 4 - t_4) \quad (3.7)$$

where t_4 is an arbitrary integer.

Next, we will solve (3.6) using Theorem 3.7. By Algorithm 2.3, we find three matrices

$$\mathbf{W} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -5 & 0 & 0 & 1 \\ 8 & -1 & -1 & -1 \\ -17 & 4 & 3 & 1 \end{pmatrix}, \mathbf{V} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -2 \\ 0 & 1 & -12 \end{pmatrix}$$

and

$$\mathbf{D} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 8 \\ 0 & 0 & 0 \end{pmatrix},$$

such that \mathbf{W} and \mathbf{V} are unimodular, \mathbf{D} is diagonal, and $\mathbf{WAV} = \mathbf{D}$. Solving the reduced system $\boldsymbol{\tau}\mathbf{D} = \mathbf{c}\mathbf{V}$, or

$$(\tau_1, -\tau_2, 8\tau_3) = (2, 8, -96),$$

we get $\tau_1 = 2, \tau_2 = -8, \tau_3 = -12$; and τ_4 remains undetermined. By Theorem 3.7, the general solution to (3.6) is

$$\mathbf{x} = \boldsymbol{\tau}\mathbf{W} = (-54 - 17\tau_4, 12 + 4\tau_4, 12 + 3\tau_4, 4 + \tau_4) \quad (3.8)$$

where τ_4 is an arbitrary integer.

It is clear that the general solutions in (3.7) and (3.8) generate the same solution-sets. In fact, we have $\tau_4 = -t_4$.

EXERCISES 3.4

1. Write an algorithm that will take an $m \times 1$ integer matrix \mathbf{A} and an integer c , and decide if the diophantine equation $\mathbf{x}\mathbf{A} = c$ has a solution. When a solution exists, you should also find the general solution. Do not call Algorithm 2.1, but you may incorporate a special version of it within your algorithm.
2. Apply the algorithm of the previous exercise to the following diophantine equations:
 - (a) $3x_1 - 3x_2 = 6$
 - (b) $10x_1 + 14x_2 = 15$
 - (c) $55x_1 - 89x_2 + 41x_3 = 17$
 - (d) $14x_1 + 21x_2 - 35x_3 + 28x_4 = -42$.
3. Prove Theorem 3.7.
4. Using Theorem 3.6, decide if each of the following systems of diophantine equations has a solution:
 - (a)
$$\begin{array}{rcl} 3x_1 & - & 3x_2 = 6 \\ 10x_1 & + & 14x_2 = 15 \end{array} \left. \right\}$$
 - (b)
$$\begin{array}{rcl} 3x_1 & + & 14x_2 = 15 \\ 55x_1 & - & 89x_2 + 41x_3 = 17 \end{array} \left. \right\}$$
 - (c)
$$\begin{array}{rcl} 10x_1 & + & 13x_2 = 15 \\ 55x_1 & - & 89x_2 + 41x_3 = 17 \end{array} \left. \right\}$$
 - (d)
$$\begin{array}{rcl} x_1 & + & 3x_2 - 2x_3 + x_4 = 5 \\ -2x_1 & - & x_2 + x_3 + 2x_4 = 8 \\ 5x_1 & + & 2x_2 - 3x_4 = 8. \end{array} \left. \right\}$$

When a solution exists, find the general solution.

5. Using Theorem 3.7, solve the systems of the previous exercise.

3.5 Equations in Two Variables

A linear diophantine equation in two variables is especially important in the dependence analysis of loops. In real programs, an

array element usually has the form $X(ai + a_0)$ where X is a one-dimensional array, I is the index variable of a loop, and a , a_0 are integer constants. Equating two subscripts of this form, we get an equation in two variables. For example, the elements $X(2i + 3)$ and $X(3j - 1)$ will represent the same memory location iff $2i + 3 = 3j - 1$ or $2i - 3j = -4$. Even for elements of a multi-dimensional array seen in a program, the subscript in each dimension tends to be a linear function of a single loop index. If the corresponding subscripts of two such array elements are matched, we will get a system of two-variable equations. Also, two distinct equations in this system often would not have any variables in common. Such a system of equations can be solved by solving separately the single two-variable equations that make up the system. We will return to this point in Section 5.6.

In this section, we will study an equation of the form

$$ai - bj = c$$

where a, b, c are integer constants and i, j are integer variables. The notation (including the minus sign between the terms) is chosen to conform to the notation of the dependence problem discussed in Chapter 5; it has no other significance.² Assume that a and b are not both zero. Then, there is a solution iff $\gcd(a, -b)$ divides c . When a solution exists, all solutions are given by a formula containing a single integer parameter which we will denote by t . (See Theorem 3.5 or its Corollary.) This parameter t can take any integral value. We are interested only in solutions that lie in a certain bounded range.³ This limited set of solutions is represented by the finite set of (integral) values of t lying between two integers τ_1 and τ_2 .

Once this set of solutions (in the given range) is computed, we want to partition the set into three subsets based on whether i is less than, greater than, or equal to j in a solution. This is equivalent to

²If we equate the subscripts of two array elements of the forms $X(ai + a_0)$ and $X(bj + b_0)$, the resulting equation will be $ai - bj = c$ where $c = b_0 - a_0$.

³Since i and j represent values of the index variable of a loop, they must lie between the loop limits.

partitioning the interval $[\tau_1, \tau_2]$ of t into three subintervals. In the case $a = b$, all solutions fall into the same category: either $i < j$, or $i > j$, or $i = j$ for *all* solutions. In the $a \neq b$ case, all three types of solutions may be present. We discuss these two cases separately in two examples (the $a \neq b$ case first), and then present the general algorithm.

Example 3.5 Consider the single loop

```
L :      do I = 0, 35
        S :      X(6I - 1) = B(I) + C(I)
        T :      D(I + 2) = X(4I + 9) + 1
        enddo
```

Suppose that we want to find all pairs of values (i, j) of the index variable I , such that the memory location written by the instance $S(i)$ of statement S is the same as the memory location read by the instance $T(j)$ of statement T . The output variable of $S(i)$ is $X(6i - 1)$ and the input variable of $T(j)$ is $X(4j + 9)$. They will represent the same memory location iff $6i - 1 = 4j + 9$, or

$$6i - 4j = 10. \quad (3.9)$$

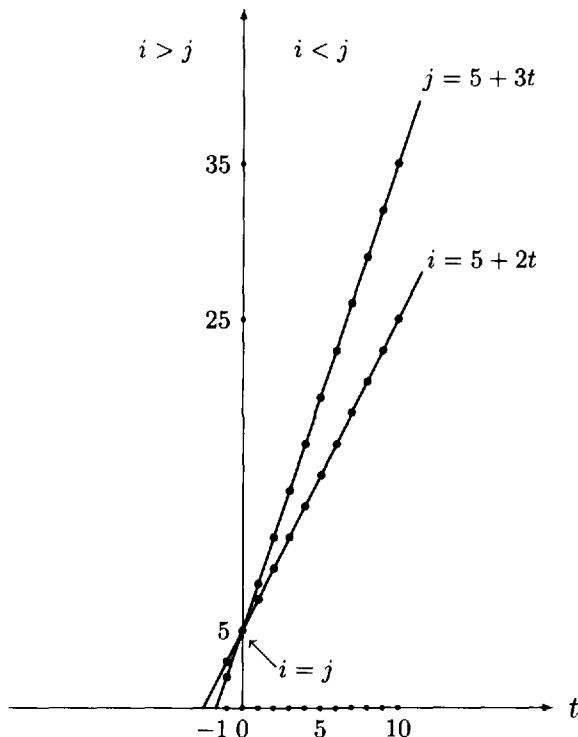
By Algorithm 2.1, find the echelon reduction of the coefficient matrix of equation (3.9):

$$\begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 6 \\ -4 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}.$$

Since the element in the first row of the echelon matrix is positive, it gives the gcd of the coefficients; that is, $\gcd(6, -4) = 2$ (Theorem 3.4). Since 2 divides 10, equation (3.9) has a solution and the general solution is

$$(i(t), j(t)) = (10/2, t) \cdot \begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} = (5 + 2t, 5 + 3t)$$

where t is an arbitrary integer (Theorem 3.5). The lines $i = 5 + 2t$ and $j = 5 + 3t$ are shown in Figure 3.1.

Figure 3.1: Graphs of the functions $i(t)$ and $j(t)$.

Since i and j are values of I , they must lie between 0 and 35, so that we have

$$\begin{aligned} 0 &\leq 5 + 2t \leq 35 \\ 0 &\leq 5 + 3t \leq 35, \end{aligned}$$

or

$$\begin{aligned} -5/2 &\leq t \leq 15 \\ -5/3 &\leq t \leq 10. \end{aligned}$$

Since t is an integer variable, its values are the integers in the range:

$$\lceil \max(-5/2, -5/3) \rceil \leq t \leq \lfloor \min(15, 10) \rfloor$$

that is, in $-1 \leq t \leq 10$. Let Ψ denote the set of all pairs of values (i, j) of I such that the output variable of $S(i)$ and the input variable of $T(j)$ represent the same memory location. We then have

$$\Psi = \{(i(t), j(t)) : i(t) = 5 + 2t, j(t) = 5 + 3t, -1 \leq t \leq 10\}.$$

Now, suppose we want to find out in which *order* the same memory location is written by the instance $S(i)$ and read by the instance $T(j)$, for a given pair (i, j) in Ψ . To determine this order, we need to break up Ψ into the following three subsets:⁴

$$\begin{aligned}\Psi_1 &= \{(i, j) \in \Psi : i < j\} \\ \Psi_{-1} &= \{(i, j) \in \Psi : i > j\} \\ \Psi_0 &= \{(i, j) \in \Psi : i = j\}.\end{aligned}$$

Note that $i(t) = j(t)$ iff $5 + 2t = 5 + 3t$, or $t = 0$. Since 0 is an integer between -1 and 10 , it is an acceptable value of t , and so

$$\Psi_0 = \{(i(0), j(0))\} = \{(5, 5)\}.$$

We have $i(t) < j(t)$ iff $5 + 2t < 5 + 3t$, or $t > 0$, or $t \geq 1$. The values of t that define Ψ_1 are precisely the integers in the intersection of the two closed intervals $[-1, 10]$ and $[1, \infty)$. Thus,

$$\begin{aligned}\Psi_1 &= \{(5 + 2t, 5 + 3t) : 1 \leq t \leq 10\} \\ &= \{(7, 8), (9, 11), \dots, (23, 32), (25, 35)\}.\end{aligned}$$

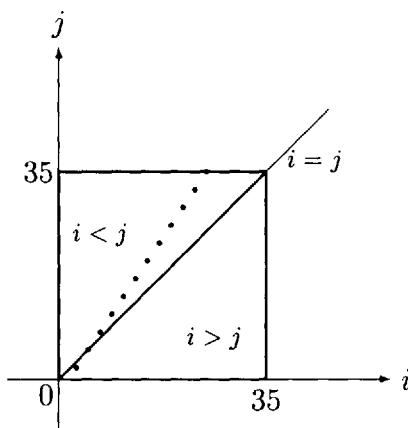
The subset Ψ_{-1} is obtained similarly:

$$\begin{aligned}\Psi_{-1} &= \{(5 + 2t, 5 + 3t) : t = -1\} \\ &= \{(3, 2)\}.\end{aligned}$$

The solution points (i, j) are plotted in Figure 3.2, which also shows the distribution of the points in the three regions $i < j$, $i > j$, and $i = j$. (The points, of course, lie on the line $6i - 4j = 10$.) The points $(t, i(t))$ and $(t, j(t))$ for $-1 \leq t \leq 10$ are shown in Figure 3.1.

It is sometimes necessary to know how close or how far apart two iterations of L could be, when one contains an instance $S(i)$ and the other an instance $T(j)$, such that both instances reference the same memory location. For (i, j) in Ψ_0 , the two instances are obviously in the same iteration of the loop. Note that $j(t) - i(t) = t$

⁴Each subscript represents the sign of $(j - i)$ in the corresponding subset.

Figure 3.2: The points (i, j) in the solution-set Ψ .

for each (i, j) in Ψ . The minimum and maximum values of $|j - i|$ for (i, j) in Ψ_1 are respectively

$$\begin{aligned}\mu_1 &= \min\{t : 1 \leq t \leq 10\} = 1 \\ \nu_1 &= \max\{t : 1 \leq t \leq 10\} = 10.\end{aligned}$$

The minimum value is attained at $t = 1$ and the maximum at $t = 10$ (Figure 3.1). The minimum and maximum values of $|j - i|$ for (i, j) in Ψ_{-1} are both equal to 1.

Example 3.6 Consider the loop:

```
L :      do I = 0,35
        S :      X(2I - 1) = B(I) + C(I)
        T :      D(I + 2) = X(2I + 9) + 1
      enddo
```

where the coefficients of I in the subscripts of the two elements of X are now the same.

An instance $S(i)$ of statement S and an instance $T(j)$ of statement T will represent the same memory location iff

$$2i - 2j = 10. \quad (3.10)$$

As before, we use Algorithm 2.1 to reduce the coefficient matrix to echelon form:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ -2 \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \end{pmatrix}.$$

By Theorem 3.4, we have $\gcd(2, -2) = |-2| = 2$. However, to apply Theorem 3.5, we need the gcd as the element in the first row of the echelon matrix (i.e., the element should be 2 instead of -2). It is easy to derive the following alternate echelon reduction of the coefficient matrix (explain):

$$\begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ -2 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}.$$

Since the gcd 2 divides the right-hand side of equation (3.10), the equation has a solution and the general solution is given by

$$(i(t), j(t)) = (10/2, t) \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} = (t, -5 + t)$$

where t is any integer (Theorem 3.5).

The graphs of the functions $i(t) = t$ and $j(t) = t - 5$ are two parallel lines, 5 units apart and with a slope of 1. Solving for t the inequalities $0 \leq t \leq 35$ and $0 \leq -5 + t \leq 35$, we get $5 \leq t \leq 35$. Hence, the set of all pairs of values (i, j) of I such that the instances $S(i)$ and $T(j)$ represent the same memory location, is

$$\begin{aligned} \Psi &= \{(i(t), j(t)) : i(t) = t, j(t) = -5 + t, 5 \leq t \leq 35\} \\ &= \{(t, -5 + t) : 5 \leq t \leq 35\} \\ &= \{(5, 0), (6, 1), \dots, (35, 30)\}. \end{aligned}$$

It is clear that we have $i(t) > j(t)$ for each $(i(t), j(t))$ in Ψ . Thus, whenever $S(i)$ and $T(j)$ reference the same memory location, the ‘write’ by $S(i)$ comes after the ‘read’ by $T(j)$. In terms of the notation introduced in Example 3.5, we have $\Psi_{-1} = \Psi$, $\Psi_1 = \emptyset$, and $\Psi_0 = \emptyset$. The iterations containing $S(i)$ and $T(j)$ are always 5 iterations apart, since for *any* (i, j) in Ψ_{-1} , the value of $|j - i|$ is 5.

The techniques of the two examples given above illustrate the steps of the following algorithm:

Algorithm 3.1 Given five integers a, b, c, p, q , this algorithm finds

- The set Ψ of all (integer) solutions (i, j) to the diophantine equation

$$ai - bj = c \quad (3.11)$$

satisfying the constraints: $p \leq i \leq q$ and $p \leq j \leq q$;

- The following subsets of Ψ :

$$\begin{aligned}\Psi_1 &= \{(i, j) \in \Psi : i < j\} \\ \Psi_{-1} &= \{(i, j) \in \Psi : i > j\} \\ \Psi_0 &= \{(i, j) \in \Psi : i = j\};\end{aligned}$$

- The minimum value μ_1 and the maximum value ν_1 of $|j - i|$ for (i, j) in Ψ_1 ; the minimum value μ_{-1} and the maximum value ν_{-1} of $|j - i|$ for (i, j) in Ψ_{-1} .

1. Initialize $\Psi, \Psi_1, \Psi_{-1}, \Psi_0$ to \emptyset .

Initialize $\mu_1, \nu_1, \mu_{-1}, \nu_{-1}$ to ‘undefined.’

2. If a and b are not both zero, then go to step 3.

[Since here $a = b = 0$, the equation degenerates to $0 = c$.]

If $c = 0$, then set

$$\begin{aligned}\Psi &\leftarrow \{(i, j) : p \leq i \leq q, p \leq j \leq q\} \\ \Psi_1 &\leftarrow \{(i, j) : p \leq i < j \leq q\} \\ \Psi_{-1} &\leftarrow \{(i, j) : p \leq j < i \leq q\} \\ \Psi_0 &\leftarrow \{(i, i) : p \leq i \leq q\}.\end{aligned}$$

If $\Psi_1 \neq \emptyset$, then set

$$\begin{aligned}\mu_1 &\leftarrow 1 \\ \nu_1 &\leftarrow q - p.\end{aligned}$$

If $\Psi_{-1} \neq \emptyset$, then set

$$\begin{aligned}\mu_{-1} &\leftarrow 1 \\ \nu_{-1} &\leftarrow q - p.\end{aligned}$$

Terminate the algorithm.

3. [Since now a and b are not both zero, their gcd is positive.]

By Algorithm 2.1, find a 2×2 unimodular matrix $\mathbf{U} = (u_{rs})$ and an echelon matrix $(s_{11}, 0)'$, such that

$$\begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix} \cdot \begin{pmatrix} a \\ -b \end{pmatrix} = \begin{pmatrix} s_{11} \\ 0 \end{pmatrix}.$$

If $s_{11} < 0$, then set

$$\begin{aligned} (u_{11}, u_{12}) &\leftarrow -(u_{11}, u_{12}) \\ s_{11} &\leftarrow -s_{11}. \end{aligned}$$

4. [Since s_{11} is positive, it gives $\gcd(a, -b)$ (Theorem 3.4).]

Set $g \leftarrow s_{11}$.

If $c \bmod g = 0$, then go to step 5.

[At this point, $c \bmod g \neq 0$, i.e., g does not divide c . Therefore equation (3.11) has no integer solution (Theorem 3.5).]

Terminate the algorithm.

5. [We have $\mathbf{U} \cdot \begin{pmatrix} a \\ -b \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix}$. By Theorem 3.5, the general solution to equation (3.11) is

$$(i, j) = (c/g, t) \cdot \mathbf{U} = (cu_{11}/g + tu_{21}, cu_{12}/g + tu_{22})$$

where t is any integer.]

Set

$$\begin{aligned} (i_0, j_0) &\leftarrow (cu_{11}/g, cu_{12}/g) \\ (i_1, j_1) &\leftarrow (u_{21}, u_{22}). \end{aligned}$$

[The general solution is then

$$(i(t), j(t)) = (i_0 + i_1 t, j_0 + j_1 t) \tag{3.12}$$

where t is any integer.]

6. [A solution $(i(t), j(t))$ will belong to Ψ iff $i(t)$ and $j(t)$ lie between p and q , i.e., iff the corresponding t satisfies

$$\begin{aligned} p - i_0 &\leq i_1 t \leq q - i_0 \\ p - j_0 &\leq j_1 t \leq q - j_0. \end{aligned}$$

We will find two integers τ_1, τ_2 such that $(i(t), j(t)) \in \Psi$ iff $\tau_1 \leq t \leq \tau_2$. These extreme values of t are found by solving the above inequalities. We need to know the signs of i_1 and j_1 before t can be isolated.]

Initialize: $\tau_1 \leftarrow -\infty, \tau_2 \leftarrow \infty$.

Select case based on the sign of i_1 :

Case ($i_1 = 0$): If $p \leq i_0 \leq q$ is false, then terminate the algorithm (Ψ is empty).

Case ($i_1 > 0$): Set

$$\begin{aligned} \tau_1 &\leftarrow \lceil (p - i_0)/i_1 \rceil \\ \tau_2 &\leftarrow \lfloor (q - i_0)/i_1 \rfloor. \end{aligned}$$

Case ($i_1 < 0$): Set

$$\begin{aligned} \tau_1 &\leftarrow \lceil (q - i_0)/i_1 \rceil \\ \tau_2 &\leftarrow \lfloor (p - i_0)/i_1 \rfloor. \end{aligned}$$

Select case based on the sign of j_1 :

Case ($j_1 = 0$): If $p \leq j_0 \leq q$ is false, then terminate the algorithm (Ψ is empty).

Case ($j_1 > 0$): Set

$$\begin{aligned} \tau_1 &\leftarrow \max(\tau_1, \lceil (p - j_0)/j_1 \rceil) \\ \tau_2 &\leftarrow \min(\tau_2, \lfloor (q - j_0)/j_1 \rfloor). \end{aligned}$$

Case ($j_1 < 0$): Set

$$\begin{aligned} \tau_1 &\leftarrow \max(\tau_1, \lceil (q - j_0)/j_1 \rceil) \\ \tau_2 &\leftarrow \min(\tau_2, \lfloor (p - j_0)/j_1 \rfloor). \end{aligned}$$

7. If $\tau_1 > \tau_2$, then terminate (Ψ is empty). Otherwise, set

$$\Psi \leftarrow \{(i_0 + i_1 t, j_0 + j_1 t) : \tau_1 \leq t \leq \tau_2\}.$$

[Since \mathbf{U} is unimodular and (i_1, j_1) is its second row, we cannot have $i_1 = j_1 = 0$. Hence, τ_1 and τ_2 are both finite at this point.]

8. If $i_1 \neq j_1$, then go to step 9.

[In this step, we consider the case $i_1 = j_1$. Equation (3.12) gives $j(t) - i(t) = j_0 - i_0$. The signs of $(j - i)$ for all solutions (i, j) are the same, so that one of the subsets $\Psi_1, \Psi_{-1}, \Psi_0$ is all of Ψ , and the other two are empty.]

Select case based on the sign of $(j_0 - i_0)$:

Case $(j_0 - i_0 = 0)$: Set $\Psi_0 \leftarrow \Psi$.

Case $(j_0 - i_0 > 0)$: Set $\Psi_1 \leftarrow \Psi$.

Case $(j_0 - i_0 < 0)$: Set $\Psi_{-1} \leftarrow \Psi$. [See Figure 3.3(a).]

If $\Psi_1 \neq \emptyset$, then set

$$\begin{aligned}\mu_1 &\leftarrow j_0 - i_0 \\ \nu_1 &\leftarrow j_0 - i_0.\end{aligned}$$

If $\Psi_{-1} \neq \emptyset$, then set

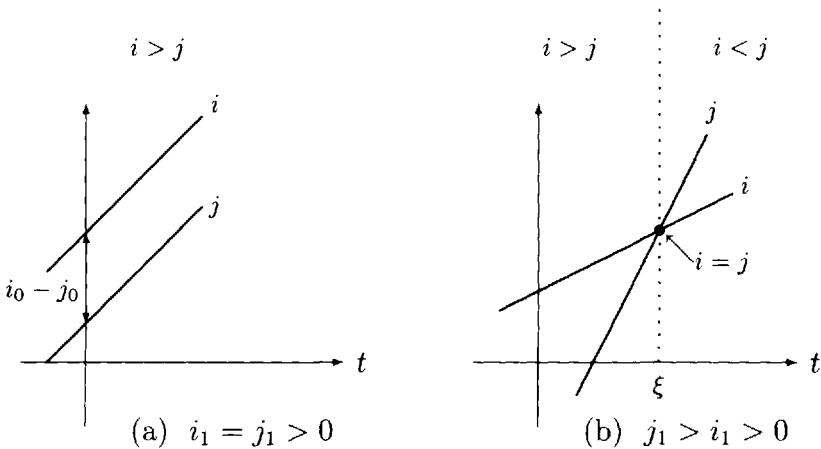
$$\begin{aligned}\mu_{-1} &\leftarrow i_0 - j_0 \\ \nu_{-1} &\leftarrow i_0 - j_0.\end{aligned}$$

Terminate the algorithm.

9. [At this point, we have $i_1 \neq j_1$. Equation (3.12) gives:

$$j(t) - i(t) = (j_0 - i_0) + (j_1 - i_1)t. \quad (3.13)$$

The unique (real) value ξ of t where $i(t) = j(t)$, is given by $(j_0 - i_0) + (j_1 - i_1)\xi = 0$. The subsets Ψ_1 and Ψ_{-1} are defined

Figure 3.3: Graphs of the functions $i(t)$ and $j(t)$.

for values of t on two sides of ξ , and ξ itself gives Ψ_0 in case it is an integer between τ_1 and τ_2 . We find Ψ_0 in this step, and Ψ_1 , Ψ_{-1} in steps 10–12.]

Set $\xi \leftarrow (i_0 - j_0)/(j_1 - i_1)$.

If ξ is an integer such that $\tau_1 \leq \xi \leq \tau_2$, then set

$$\Psi_0 \leftarrow \{(i_0 + i_1\xi, j_0 + j_1\xi)\}.$$

10. [Equation (3.13) shows that

$$\begin{aligned} i(t) < j(t) &\text{ iff } (j_1 - i_1)t > i_0 - j_0 \\ i(t) > j(t) &\text{ iff } (j_1 - i_1)t < i_0 - j_0. \end{aligned}$$

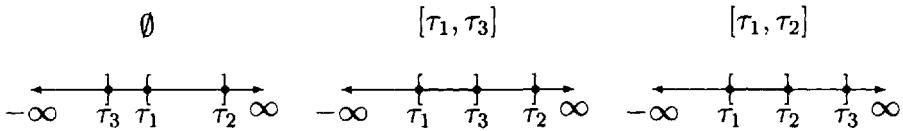
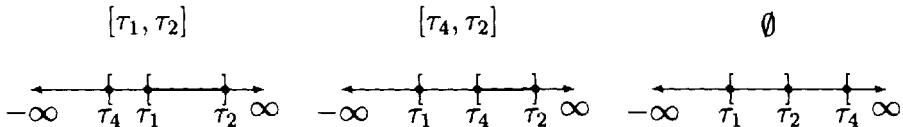
There are two cases:

Case ($j_1 - i_1 > 0$): (See Figure 3.3(b).)

$$\begin{aligned} i(t) < j(t) &\text{ iff } t > \xi \\ i(t) > j(t) &\text{ iff } t < \xi. \end{aligned}$$

Case ($j_1 - i_1 < 0$):

$$\begin{aligned} i(t) < j(t) &\text{ iff } t < \xi \\ i(t) > j(t) &\text{ iff } t > \xi. \end{aligned}$$

Figure 3.4: The interval $[\tau_1, \tau_5] = [\tau_1, \tau_2] \cap (-\infty, \tau_3)$.Figure 3.5: The interval $[\tau_6, \tau_2] = [\tau_1, \tau_2] \cap [\tau_4, \infty)$.

Note that $[\xi - 1]$ is the largest integer strictly smaller than ξ , and $[\xi + 1]$ is the smallest integer strictly greater than ξ . Since t is an integer, we can replace the condition $t < \xi$ by $t \leq [\xi - 1]$, and the condition $t > \xi$ by $t \geq [\xi + 1]$.

Set $\tau_3 \leftarrow [\xi - 1]$ and $\tau_4 \leftarrow [\xi + 1]$.

- 11.** [Find τ_5 and τ_6 such that

$$\begin{aligned} [\tau_1, \tau_5] &= [\tau_1, \tau_2] \cap (-\infty, \tau_3] \\ [\tau_6, \tau_2] &= [\tau_1, \tau_2] \cap [\tau_4, \infty). \end{aligned}$$

See Figures 3.4 and 3.5.]

Set $\tau_5 \leftarrow \min(\tau_2, \tau_3)$ and $\tau_6 \leftarrow \max(\tau_1, \tau_4)$.

- 12.** [Let $j_1 - i_1 > 0$. Then, the integers from τ_6 to τ_2 give the values of t for which a solution $(i(t), j(t))$ in Ψ satisfies $i(t) < j(t)$. Also, the integers from τ_1 to τ_5 give the values of t for which a solution $(i(t), j(t))$ in Ψ satisfies $i(t) > j(t)$. When $j_1 - i_1 < 0$, these ranges are switched.]

Select case based on the sign of $(j_1 - i_1)$:

Case ($j_1 - i_1 > 0$): If $\tau_6 \leq \tau_2$, then set

$$\Psi_1 \leftarrow \{(i_0 + i_1 t, j_0 + j_1 t) : \tau_6 \leq t \leq \tau_2\}.$$

If $\tau_1 \leq \tau_5$, then set

$$\Psi_{-1} \leftarrow \{(i_0 + i_1 t, j_0 + j_1 t) : \tau_1 \leq t \leq \tau_5\}.$$

Case ($j_1 - i_1 < 0$): If $\tau_1 \leq \tau_5$, then set

$$\Psi_1 \leftarrow \{(i_0 + i_1 t, j_0 + j_1 t) : \tau_1 \leq t \leq \tau_5\}.$$

If $\tau_6 \leq \tau_2$, then set

$$\Psi_{-1} \leftarrow \{(i_0 + i_1 t, j_0 + j_1 t) : \tau_6 \leq t \leq \tau_2\}.$$

13. [Finally, we find the extreme values of $|j - i|$ for (i, j) in Ψ_1 , and for (i, j) in Ψ_{-1} . We apply Lemma 3.2 to:
 $|j - i| = j - i = (j_0 - i_0) + (j_1 - i_1)t$ for (i, j) in Ψ_1 ,
 $|j - i| = i - j = (i_0 - j_0) + (i_1 - j_1)t$ for (i, j) in Ψ_{-1} . The superscripts '+' and '-' represent positive and negative parts.]

If Ψ_1 is nonempty and of the form

$$\{(i, j) : i = i_0 + i_1 t, j = j_0 + j_1 t, \alpha \leq t \leq \beta\}, \quad (3.14)$$

then set

$$\begin{aligned}\mu_1 &\leftarrow (j_0 - i_0) + (j_1 - i_1)^+ \alpha - (j_1 - i_1)^- \beta \\ \nu_1 &\leftarrow (j_0 - i_0) + (j_1 - i_1)^+ \beta - (j_1 - i_1)^- \alpha.\end{aligned}$$

If Ψ_{-1} is nonempty and of the form (3.14), then set

$$\begin{aligned}\mu_{-1} &\leftarrow (i_0 - j_0) + (i_1 - j_1)^+ \alpha - (i_1 - j_1)^- \beta \\ \nu_{-1} &\leftarrow (i_0 - j_0) + (i_1 - j_1)^+ \beta - (i_1 - j_1)^- \alpha.\end{aligned}$$

14. Terminate the algorithm. □

We should emphasize the difference between the two cases where i_1 and j_1 are or are not equal. Ignore the constraints on i and j for now. If $i_1 = j_1$, then the lines representing i and j are parallel, so that either they are identical, or one lies above the other (Figure 3.3(a)). The difference $(j - i)$ in this case is a constant. If $i_1 \neq j_1$, then the lines intersect, so that as we move from $-\infty$ to ∞ , the sign of $(j - i)$ is fixed for a while, and then changes exactly once (Figure 3.3(b)). In this case, the values of the difference $(j - i)$ form an infinite set, unbounded above and below. As Exercise 1 shows, $i_1 = j_1$ iff $a = b$. Thus, step 8 of the algorithm really deals with the case $a = b \neq 0$, and steps 9–14 with the case $a \neq b$.

When nonempty, the set Ψ has $(\tau_2 - \tau_1 + 1)$ solutions. The number of solutions in each of the subsets $\Psi_1, \Psi_{-1}, \Psi_0$ can be easily computed in each of the cases discussed in the algorithm.

EXERCISES 3.5

1. Show that in Algorithm 3.1, (i_1, j_1) is either $(b/g, a/g)$, or $(-b/g, -a/g)$.
2. Consider all 2×2 unimodular matrices \mathbf{U} such that $\mathbf{U} \cdot (a, -b)' = (g, 0)'$. Show that if \mathbf{U} and \mathbf{V} are two matrices with this property, then

$$\begin{aligned}(v_{11}, v_{12}) &= (u_{11}, u_{12}) + T(b/g, a/g) \\ (v_{21}, v_{22}) &= \pm(u_{21}, u_{22})\end{aligned}$$

for some integer T .

(Any such matrix \mathbf{U} can be used in Algorithm 3.1, not just the one produced by Algorithm 2.1. The solution-sets are independent of the choice of a particular matrix, although the formulas that describe the sets may depend on that choice.)

3. Let $a \neq b$. Show that equation (3.11) has an integer solution (i, j) with $i = j$, iff $(a - b)$ divides c . Show also that if this condition holds, then the only such solution is (γ, γ) where $\gamma = c/(a - b) = i(\xi) = j(\xi)$. ($i(t), j(t), \xi$ are the same as in Algorithm 3.1.)
4. Show that when $a = b \neq 0$ and a divides c , the general solution to equation (3.11) can be written in the form

$$\{(c/a + t, t) : -\infty < t < \infty\}.$$

What is the matrix \mathbf{U} in this case?

5. For each special equation, give the condition under which it has a solution, and find the general solution when solutions exist:

- (a) $ai - bj = 0$
 (b) $ai - bj = 1$
 (c) $ai - bj = g$
 (d) $ai + aj = c$
 (e) $i - bj = c$
 (f) $ai + j = c.$
6. Apply Algorithm 3.1 to each of the following equations to find the sets Ψ , Ψ_1 , Ψ_{-1} and Ψ_0 . Assume that $p = -200$ and $q = 200$. If the set Ψ_1 or Ψ_{-1} is nonempty, find the extreme values of $|j - i|$ in that set.
- (a) $3i - 3j = 6$
 (b) $3i - 7j = 6$
 (c) $-3i + 19j = 2$
 (d) $10i - 14j = 10$
 (e) $10i - 14j = 4$
 (f) $3i + 3j = 6$
 (g) $22i - 33j = 10$
 (h) $5i - 2j = -3$
 (i) $55i - 89j = 12$
 (j) $2i + 3j = 46$
 (k) $-3i - 2j = 6.$

3.6 Fourier's Method of Elimination

Elimination of variables is a powerful tool for solving a system of linear equations. The French mathematician Fourier designed a method for solving a system of linear inequalities using variable elimination (circa 1827). His method has been rediscovered several times over the years by other mathematicians.⁵ Although Fourier was not concerned with optimizing any objective function, his method can be adapted to solve the linear optimization problem.

⁵This method is currently referred to as the *Fourier-Motzkin Elimination Method*. For simplicity, we will use only Fourier's name.

In this section, we will only consider the original Fourier method for solving linear inequalities.

As we pointed out in Section 3.1, this method decides if there is a real solution to a system of linear inequalities with real coefficients. We will assume here that the coefficients are rationals and that the arithmetic is exact. First, we illustrate the method by a simple example. Then, a discussion of the general method is given, which is then followed by the formal algorithm. Finally, there is an example that shows how the Fourier method can be applied to compute the new limits of a loop nest after a transformation.

Example 3.7 Let us solve the system of linear inequalities:

$$\left. \begin{array}{rcl} 2x_1 - 11x_2 & \leq & 3 \\ -3x_1 + 2x_2 & \leq & -5 \\ x_1 + 3x_2 & \leq & 4 \\ -2x_1 & \leq & -3. \end{array} \right\}$$

Rearrange the inequalities so that the ones with a positive coefficient for x_2 are at the top; they are followed by the ones where this coefficient is negative, and then come the inequalities where x_2 is absent:

$$\left. \begin{array}{rcl} -3x_1 + 2x_2 & \leq & -5 \\ x_1 + 3x_2 & \leq & 4 \\ 2x_1 - 11x_2 & \leq & 3 \\ -2x_1 & \leq & -3. \end{array} \right\} \quad (3.15)$$

Divide the first three inequalities by their corresponding coefficient of x_2 :

$$\left. \begin{array}{rcl} -\frac{3}{2}x_1 + x_2 & \leq & -\frac{5}{2} \\ \frac{1}{3}x_1 + x_2 & \leq & \frac{4}{3} \\ -\frac{2}{11}x_1 + x_2 & \geq & -\frac{3}{11}. \end{array} \right\}$$

Note how the direction of the third inequality is reversed after division, as the coefficient of x_2 in it was negative. Isolating x_2 in this

system, we get

$$\left. \begin{aligned} x_2 &\leq \frac{3}{2}x_1 - \frac{5}{2} \\ x_2 &\leq -\frac{1}{3}x_1 + \frac{4}{3} \\ \frac{2}{11}x_1 - \frac{3}{11} &\leq x_2. \end{aligned} \right\}$$

Thus, when x_1 is given, x_2 must satisfy

$$b_2(x_1) \leq x_2 \leq B_2(x_1) \quad (3.16)$$

where

$$\begin{aligned} b_2(x_1) &= \frac{2}{11}x_1 - \frac{3}{11} \\ B_2(x_1) &= \min\left(\frac{3}{2}x_1 - \frac{5}{2}, -\frac{1}{3}x_1 + \frac{4}{3}\right). \end{aligned}$$

We now eliminate x_2 by setting each of its lower bounds less than or equal to each of its upper bounds:

$$\left. \begin{aligned} \frac{2}{11}x_1 - \frac{3}{11} &\leq \frac{3}{2}x_1 - \frac{5}{2} \\ \frac{2}{11}x_1 - \frac{3}{11} &\leq -\frac{1}{3}x_1 + \frac{4}{3}. \end{aligned} \right\}$$

Simplify these inequalities and bring in the last member of (3.15):

$$\left. \begin{aligned} -\frac{29}{22}x_1 &\leq -\frac{49}{22} \\ \frac{17}{33}x_1 &\leq \frac{53}{33} \\ -2x_1 &\leq -3. \end{aligned} \right\}$$

There is no inequality where x_1 is absent. We rearrange the system so that the single inequality with a positive coefficient for x_1 goes to the top:

$$\left. \begin{aligned} \frac{17}{33}x_1 &\leq \frac{53}{33} \\ -\frac{29}{22}x_1 &\leq -\frac{49}{22} \\ -2x_1 &\leq -3. \end{aligned} \right\}$$

Divide each inequality by the corresponding coefficient of x_1 (and reverse the direction if that coefficient is negative):

$$\left. \begin{aligned} x_1 &\leq \frac{53}{17} \\ x_1 &\geq \frac{49}{29} \\ x_1 &\geq \frac{3}{2}. \end{aligned} \right\}$$

The minimum value of x_1 is $\max(49/29, 3/2)$ or $49/29$, and the maximum value is $53/17$. Since the range

$$\frac{49}{29} \leq x_1 \leq \frac{53}{17} \quad (3.17)$$

is nonempty, the original system of inequalities has a solution. All solutions have the form (x_1, x_2) where x_1 is a real number in the range (3.17), and x_2 satisfies (3.16) for that value of x_1 .

We start with a system of n linear inequalities in m real variables x_1, x_2, \dots, x_m :

$$a_{1j}x_1 + a_{2j}x_2 + \cdots + a_{mj}x_m \leq c_j \quad (1 \leq j \leq n)$$

or

$$\sum_{i=1}^m a_{ij}x_i \leq c_j \quad (1 \leq j \leq n), \quad (3.18)$$

where the coefficients a_{ij} and c_j are rational constants. To solve this system, we eliminate the variables one at a time in the order: x_m, x_{m-1}, \dots, x_1 . Note that for rational numbers p and q , the notation p/q will represent the exact result of division of p by q .

Rearrange the system (3.18) such that the inequalities with a_{mj} positive come first, followed by the inequalities with a_{mj} negative, and then those with $a_{mj} = 0$. Rename the coefficients to make a_{ij} the coefficient of x_i in the j^{th} inequality (counted from top) of the current system. Find integers n_1 and n_2 , such that $0 \leq n_1 \leq n_2 \leq n$ and

$$a_{mj} \begin{cases} > 0 & \text{if } 1 \leq j \leq n_1 \\ < 0 & \text{if } n_1 + 1 \leq j \leq n_2 \\ = 0 & \text{if } n_2 + 1 \leq j \leq n. \end{cases}$$

For $1 \leq j \leq n_2$, divide the j^{th} inequality by a_{mj} to get

$$\sum_{i=1}^{m-1} t_{ij}x_i + x_m \leq q_j \quad (1 \leq j \leq n_1)$$

$$\sum_{i=1}^{m-1} t_{ij}x_i + x_m \geq q_j \quad (n_1 + 1 \leq j \leq n_2)$$

where

$$\left. \begin{array}{rcl} t_{ij} & = & a_{ij}/a_{mj} \\ q_j & = & c_j/a_{mj} \end{array} \right\} \quad (1 \leq i \leq m-1, 1 \leq j \leq n_2).$$

Note that $(-\sum_{i=1}^{m-1} t_{ij}x_i + q_j)$ is a lower bound for x_m for $n_1 + 1 \leq j \leq n_2$, and an upper bound for x_m for $1 \leq j \leq n_1$. Let b_m denote the maximum of the lower bounds, and B_m the minimum of the upper bounds:

$$\begin{aligned} b_m(x_1, x_2, \dots, x_{m-1}) &= \max_{n_1+1 \leq j \leq n_2} \left(-\sum_{i=1}^{m-1} t_{ij}x_i + q_j \right) \\ B_m(x_1, x_2, \dots, x_{m-1}) &= \min_{1 \leq j \leq n_1} \left(-\sum_{i=1}^{m-1} t_{ij}x_i + q_j \right). \end{aligned}$$

We define b_m to be $-\infty$ if $n_2 = n_1$ (no lower bound), and B_m to be ∞ if $n_1 = 0$ (no upper bound). In terms of x_1, x_2, \dots, x_{m-1} , the range of x_m is

$$b_m(x_1, x_2, \dots, x_{m-1}) \leq x_m \leq B_m(x_1, x_2, \dots, x_{m-1}). \quad (3.19)$$

Relabel the $(n_2 - n_1)$ lower bounds of x_m as $(-\sum_{i=1}^{m-1} t_{it}x_i + q_\ell)$, $n_1 + 1 \leq \ell \leq n_2$, and the n_1 upper bounds as $(-\sum_{i=1}^{m-1} t_{ik}x_i + q_k)$, $1 \leq k \leq n_1$. To eliminate x_m , we set each lower bound less than or equal to each upper bound, and get the $n_1(n_2 - n_1)$ inequalities:

$$\sum_{i=1}^{m-1} (t_{ik} - t_{it})x_i \leq q_k - q_\ell \quad (1 \leq k \leq n_1, n_1 + 1 \leq \ell \leq n_2).$$

Add to this list the $n - n_2$ inequalities we originally had in which the coefficient of x_m was zero.

Now we have a system of $(n - n_2 + n_1(n_2 - n_1))$ inequalities in $(m - 1)$ variables. The original system (3.18) in x_1, x_2, \dots, x_m has a solution iff this new system in x_1, x_2, \dots, x_{m-1} has a solution. A real m -vector (x_1, x_2, \dots, x_m) is a solution to the original system, iff the real $(m - 1)$ -vector $(x_1, x_2, \dots, x_{m-1})$ is a solution to this new system and x_m satisfies (3.19).

Suppose the coefficients a_{mj} of x_m in the original system were all positive ($n_1 = n_2 = n$). Then, each inequality would yield an

upper bound of x_m , but there would be no lower bound (B_m is finite, $b_m = -\infty$). Elimination of x_m would not produce any inequalities ($n - n_2 + n_1(n_2 - n_1) = 0$). In this case, the original system is satisfied by any real vector (x_1, x_2, \dots, x_m) , where x_1, x_2, \dots, x_{m-1} are chosen arbitrarily, and then an x_m satisfying $x_m \leq B_m$ is picked. We will have a similar situation when the coefficients a_{mj} of x_m are all negative in the given system of inequalities. (What happens if all the coefficients a_{mj} are zero?)

If elimination of x_m produces new inequalities, then we eliminate x_{m-1} from them, after finding the range for x_{m-1} in terms of x_1, x_2, \dots, x_{m-2} . If we keep repeating this elimination process and there are always inequalities left after an elimination, then we will eventually get a system of inequalities in one variable, that is equivalent to the original system. From this final set, we will get the range of x_1 :

$$b_1 \leq x_1 \leq B_1$$

where the ‘functions’ b_1 and B_1 are rational constants or $\pm\infty$. There may also be some inequalities where x_1 is absent. They would be of the form: $0 \leq q_j$ where q_j is a rational constant. If $b_1 > B_1$, or if there is an inequality of the form $0 \leq q_j$ with $q_j < 0$, then there is no solution to (3.18). Otherwise, the solution-set consists of all real vectors (x_1, x_2, \dots, x_m) where

$$b_i(x_1, x_2, \dots, x_{i-1}) \leq x_i \leq B_i(x_1, x_2, \dots, x_{i-1}) \quad (1 \leq i \leq m).$$

The steps of the algorithm are formally stated below:

Algorithm 3.2 (Fourier Elimination) Given an $m \times n$ rational matrix $\mathbf{A} = (a_{ij})$ and a rational n -vector $\mathbf{c} = (c_1, c_2, \dots, c_n)$, this algorithm decides if there is a real m -vector $\mathbf{x} = (x_1, x_2, \dots, x_m)$ satisfying the system of n inequalities:⁶

$$\mathbf{x}\mathbf{A} \leq \mathbf{c}. \tag{3.20}$$

⁶If $\mathbf{u} = (u_1, u_2)$ and $\mathbf{v} = (v_1, v_2)$, then $\mathbf{u} < \mathbf{v}$ means $u_1 < v_1$ and $u_2 < v_2$. In contrast, recall that $\mathbf{u} \prec \mathbf{v}$ means either $u_1 < v_1$, or $(u_1 = v_1 \text{ and } u_2 < v_2)$.

When this system has a solution, the algorithm also finds a description of the solution-set in the form:

$$\left. \begin{array}{l} b_m(x_1, x_2, \dots, x_{m-1}) \leq x_m \leq B_m(x_1, x_2, \dots, x_{m-1}) \\ b_{m-1}(x_1, x_2, \dots, x_{m-2}) \leq x_{m-1} \leq B_{m-1}(x_1, x_2, \dots, x_{m-2}) \\ \vdots \\ b_1 \leq x_1 \leq B_1 \end{array} \right\}$$

where $b_i(x_1, x_2, \dots, x_{i-1})$ is $-\infty$ or the maximum of a set of linear functions with rational coefficients, and $B_i(x_1, x_2, \dots, x_{i-1})$ is ∞ or the minimum of a set of linear functions with rational coefficients, $1 \leq i \leq m$.

We work with an $r \times s$ rational matrix (t_{ij}) and a rational s -vector (q_1, q_2, \dots, q_s) , where the values of r and s vary during the course of the algorithm. The variable that is currently being eliminated is always x_r .

1. Set

$$\begin{aligned} r &\leftarrow m \\ s &\leftarrow n \\ t_{ij} &\leftarrow a_{ij} \quad (1 \leq i \leq r, 1 \leq j \leq s) \\ q_j &\leftarrow c_j \quad (1 \leq j \leq s). \end{aligned}$$

[We have a system of s inequalities in r variables, of the form

$$\sum_{i=1}^r t_{ij} x_i \leq q_j \quad (1 \leq j \leq s).]$$

2. [Rearrange the inequalities so that the ones with t_{rj} positive come first, then those with t_{rj} negative, and finally the ones with t_{rj} equal to zero.]

Rename the coefficients t_{ij} and find integers n_1, n_2 , such that

$$t_{rj} \left\{ \begin{array}{ll} > 0 & \text{if } 1 \leq j \leq n_1 \\ < 0 & \text{if } n_1 + 1 \leq j \leq n_2 \\ = 0 & \text{if } n_2 + 1 \leq j \leq s. \end{array} \right.$$

3. Set

$$\left. \begin{array}{l} t_{ij} \leftarrow t_{ij}/t_{rj} \\ q_j \leftarrow q_j/t_{rj} \end{array} \right\} \quad (1 \leq i \leq r-1, 1 \leq j \leq n_2).$$

[The inequalities become:

$$\begin{aligned} t_{1j}x_1 + t_{2j}x_2 + \cdots + t_{(r-1)j}x_{r-1} + x_r &\leq q_j \quad (1 \leq j \leq n_1) \\ t_{1j}x_1 + t_{2j}x_2 + \cdots + t_{(r-1)j}x_{r-1} + x_r &\geq q_j \quad (n_1 + 1 \leq j \leq n_2) \\ t_{1j}x_1 + t_{2j}x_2 + \cdots + t_{(r-1)j}x_{r-1} &\leq q_j \quad (n_2 + 1 \leq j \leq s). \end{aligned}$$

4. If $n_2 > n_1$, then define

$$b_r(x_1, x_2, \dots, x_{r-1}) = \max_{n_1+1 \leq j \leq n_2} \left(- \sum_{i=1}^{r-1} t_{ij}x_i + q_j \right),$$

else define $b_r = -\infty$.

If $n_1 > 0$, then define

$$B_r(x_1, x_2, \dots, x_{r-1}) = \min_{1 \leq j \leq n_1} \left(- \sum_{i=1}^{r-1} t_{ij}x_i + q_j \right),$$

else define $B_r = \infty$.

5. If $r > 1$, then go to step 6.

[Here we have $r = 1$, so that we are down to the last variable. The value of the sum $\sum_{i=1}^{r-1} t_{ij}x_i$ is to be taken as zero in the above formulas.]

If $b_1 > B_1$, or if $q_j < 0$ for some j in $n_2 + 1 \leq j \leq s$, then the system (3.20) has no solution. Otherwise the solution-set consists of all real vectors (x_1, x_2, \dots, x_m) such that

$$b_i(x_1, x_2, \dots, x_{i-1}) \leq x_i \leq B_i(x_1, x_2, \dots, x_{i-1}) \quad (1 \leq i \leq m).$$

Terminate the algorithm.

6. [To eliminate x_r , we set each lower bound $(-\sum_{i=1}^{r-1} t_{i\ell}x_i + q_\ell)$, $n_1 + 1 \leq \ell \leq n_2$, less than or equal to each upper bound $(-\sum_{i=1}^{r-1} t_{ik}x_i + q_k)$, $1 \leq k \leq n_1$. This creates $n_1(n_2 - n_1)$ inequalities. Add to this list the $s - n_2$ inequalities of step 2 in which the coefficient of x_r was zero. The total number of inequalities after the elimination of x_r is denoted by s' .]

Set $s' \leftarrow s - n_2 + n_1(n_2 - n_1)$.

If $s' > 0$, then go to step 7.

[There are no more inequalities ($s' = 0$).]

The system (3.20) has a solution. The solution-set consists of all real vectors (x_1, x_2, \dots, x_m) , where $x_{r-1}, x_{r-2}, \dots, x_1$ are chosen arbitrarily, and x_m, x_{m-1}, \dots, x_r satisfy the constraints:

$$b_i(x_1, x_2, \dots, x_{i-1}) \leq x_i \leq B_i(x_1, x_2, \dots, x_{i-1}) \quad (m \geq i \geq r).$$

Terminate the algorithm.

7. [There are now s' inequalities in $r-1$ variables x_1, x_2, \dots, x_{r-1} .]

Turn the system of inequalities

$$\begin{aligned} \sum_{i=1}^{r-1} (t_{ik} - t_{i\ell})x_i &\leq q_k - q_\ell \quad (1 \leq k \leq n_1, n_1 + 1 \leq \ell \leq n_2) \\ \sum_{i=1}^{r-1} t_{ij}x_i &\leq q_j \quad (n_2 + 1 \leq j \leq s) \end{aligned}$$

into a new system of s inequalities in r variables, of the form

$$\sum_{i=1}^r t_{ij}x_i \leq q_j \quad (1 \leq j \leq s)$$

by setting $r \leftarrow r - 1$, $s \leftarrow s'$, and defining a new $r \times s$ matrix (t_{ij}) and a new s -vector (q_1, q_2, \dots, q_s) .

Go to step 2. □

Even when the matrix \mathbf{A} and the vector \mathbf{c} are integral, Algorithm 3.2 decides if there is a *real* solution to the system $\mathbf{x}\mathbf{A} \leq \mathbf{c}$. In general, if a real solution exists and we want an integer solution, then we have to search the solution-set for an integer vector. If there is no integer between b_1 and B_1 , then there are no integer solutions. Otherwise, for each integer z_1 in $b_1 \leq z_1 \leq B_1$, we look for integers z_2 in the range $b_2(z_1) \leq z_2 \leq B_2(z_1)$, and so on. (See exercises 1 and 2.) The set of all integer solutions (z_1, z_2, \dots, z_m) is given by the inequalities

$$\left. \begin{aligned} \lceil b_m(z_1, z_2, \dots, z_{m-1}) \rceil &\leq z_m & \leq \lfloor B_m(z_1, z_2, \dots, z_{m-1}) \rfloor \\ \lceil b_{m-1}(z_1, z_2, \dots, z_{m-2}) \rceil &\leq z_{m-1} & \leq \lfloor B_{m-1}(z_1, z_2, \dots, z_{m-2}) \rfloor \\ &\vdots \\ \lceil b_1 \rceil &\leq z_1 & \leq \lfloor B_1 \rfloor \end{aligned} \right\}$$

Thus, in Example 3.7, the set of all integer vectors (z_1, z_2) satisfying the four inequalities is given by

$$\begin{aligned} \lceil \frac{2}{11}z_1 - \frac{3}{11} \rceil &\leq z_2 & \leq \lfloor \min\left(\frac{3}{2}z_1 - \frac{5}{2}, -\frac{1}{3}z_1 + \frac{4}{3}\right) \rfloor \\ 2 &\leq z_1 & \leq 3 \end{aligned}$$

since $\lceil 49/29 \rceil = 2$ and $\lfloor 53/17 \rfloor = 3$. We leave it to the reader to show that this set is empty, so that there are real solutions to the system, but no integer solutions.

It is worth noting here that a system of the form

$$\mathbf{x}\mathbf{A} \geq \mathbf{c}$$

can be solved in the same way.

The Fourier method is a useful tool for finding the new loop limits when a loop nest is changed by a loop transformation. This is illustrated by the following example and by some of the exercises at the end of the section. One should keep in mind, however, that nothing is assumed in these problems about the possible equivalence of the transformed program to the original loop nest.

Example 3.8 Consider a loop nest of the form

```

do  $I_1 = 1, 100$ 
  do  $I_2 = I_1 + 1, 100$ 
    do  $I_3 = I_2, I_1 + I_2$ 
       $H(I_1, I_2, I_3)$ 
    enddo
  enddo
enddo

```

Suppose we want to permute the loops such that the I_3 -loop becomes the outermost loop and the I_2 -loop the innermost loop. After the transformation, the new index variables (counted from the outermost loop inward) will then be K_1, K_2, K_3 , where $(K_1, K_2, K_3) = (I_3, I_1, I_2)$. The limits of the new loops must be such that K_1 lies between two constants, K_2 between two functions of K_1 , and K_3 between two functions of K_1 and K_2 . We will apply Algorithm 3.2 to find these limits.

The index variables I_1, I_2, I_3 in the given loop nest satisfy the following constraints:

$$\left. \begin{array}{l} 1 \leq I_1 \leq 100 \\ I_1 + 1 \leq I_2 \leq 100 \\ I_2 \leq I_3 \leq I_1 + I_2. \end{array} \right\} \quad (3.21)$$

Rewriting these inequalities in terms of K_1, K_2, K_3 , we get (after rearrangement):

$$\left. \begin{array}{l} K_3 \leq 100 \\ -K_1 + K_3 \leq 0 \\ K_2 - K_3 \leq -1 \\ K_1 - K_2 - K_3 \leq 0 \\ -K_2 \leq -1 \\ K_2 \leq 100. \end{array} \right\}$$

The first four inequalities give the bounds on K_3 :

$$\max(K_2 + 1, K_1 - K_2) \leq K_3 \leq \min(100, K_1). \quad (3.22)$$

Eliminating K_3 , we get the system (after simplification and rearrangement):

$$\left. \begin{array}{l} K_2 \leq 100 \\ K_2 \leq 99 \\ -K_1 + K_2 \leq -1 \\ K_1 - K_2 \leq 100 \\ -K_2 \leq 0 \\ -K_2 \leq -1. \end{array} \right\}$$

This system gives the range of K_2 :

$$\max(K_1 - 100, 0, 1) \leq K_2 \leq \min(100, 99, K_1 - 1)$$

or (this simplification step is not shown in the algorithm)

$$\max(K_1 - 100, 1) \leq K_2 \leq \min(99, K_1 - 1). \quad (3.23)$$

Eliminating K_2 , we get (after simplification and rearrangement)

$$\left. \begin{array}{l} K_1 \leq 199 \\ -K_1 \leq -2 \\ 0 \leq 98 \\ 0 \leq 99. \end{array} \right\}$$

The range of K_1 is:

$$2 \leq K_1 \leq 199. \quad (3.24)$$

The transformed loop nest consists of three loops with index variables K_1, K_2, K_3 , where the ranges of the variables are given by (3.24), (3.23), and (3.22). As it is customary to use the same index variables after a loop permutation, we will write (I_3, I_1, I_2) in place of (K_1, K_2, K_3) , and present the transformed loop nest as

```

do  $I_3 = 2, 199$ 
  do  $I_1 = \max(1, I_3 - 100), \min(99, I_3 - 1)$ 
    do  $I_2 = \max(I_1 + 1, I_3 - I_1), \min(100, I_3)$ 
       $H(I_1, I_2, I_3)$ 
    enddo
  enddo
enddo
```

EXERCISES 3.6

1. Show that each system of inequalities given below has infinitely many real solutions, but no integer solution:

$$(a) \quad \left. \begin{array}{rcl} -x_1 & + & x_2 \leq 0 \\ 2x_1 & \leq & 5 \\ - & 10x_2 & \leq -23 \end{array} \right\}$$

$$(b) \quad \left. \begin{array}{rcl} -x_1 & \leq & 0 \\ x_1 & \leq & 100 \\ x_1 & - & 600x_2 \leq -200 \\ x_1 & + & 600x_2 \leq 300. \end{array} \right\}$$

Draw the solution-set on the x_1x_2 -plane in each case.

2. Suppose Algorithm 3.2 shows that there is a real solution to a system of inequalities of the form

$$x_{k(j)} - x_{\ell(j)} \leq c_j \quad (1 \leq j \leq n)$$

where $1 \leq k(j) \leq m$ and $1 \leq \ell(j) \leq m$ for each j , and the c_j 's are all integers. Explain why we can conclude that there must also be an integer solution in this case.

3. Using Algorithm 3.2, decide if each of the following systems of inequalities has a real solution. When a solution exists, express the solution-set in terms of the functions b_i and B_i :

$$(a) \quad \left. \begin{array}{rcl} 2x_1 & + & 3x_2 \leq 100 \\ 3x_1 & - & x_2 \leq 2 \\ 5x_1 & \leq & 40 \end{array} \right\}$$

$$(b) \quad \left. \begin{array}{rcl} x_1 & + & x_2 - 3x_3 \leq 1 \\ -x_1 & + & 2x_2 - x_3 \leq 1 \\ 3x_1 & + & x_2 - 2x_3 \leq 1 \end{array} \right\}$$

$$(c) \quad \left. \begin{array}{rcl} 2x_1 & + & 3x_2 - x_3 \leq 3 \\ -x_1 & + & x_2 - x_3 \leq 2 \\ 2x_1 & + & x_2 + x_3 \leq 4 \\ -x_1 & & \leq 0 \\ - & x_2 & \leq 0 \\ - & x_3 & \leq 0. \end{array} \right\}$$

4. Describe the set of integer vectors in the solution-set of each system in Exercise 3.

5. Find the new loop limits when the I_1 and I_2 loops are interchanged in the following program:

```
do  $I_1 = 0, 100$ 
  do  $I_2 = 0, I_1 + 5$ 
     $H(I_1, I_2)$ 
  enddo
enddo
```

6. Consider a double loop:

```
do  $I_1 = 5, 100$ 
  do  $I_2 = 16, 80$ 
     $H(I_1, I_2)$ 
  enddo
enddo
```

Suppose that the index variables are changed by a unimodular matrix

$$\mathbf{U} = \begin{pmatrix} 2 & 1 \\ -1 & 0 \end{pmatrix},$$

that is, (I_1, I_2) is mapped to $(K_1, K_2) = (2I_1 - I_2, I_1)$. The given double loop will change into a double loop of the form

```
do  $K_1 = b_1, B_1$ 
  do  $K_2 = b_2(K_1), B_2(K_1)$ 
     $H_U(K_1, K_2)$ 
  enddo
enddo
```

Use Algorithm 3.2 to find the new loop limits. (See Example 2.1.)

7. Repeat the previous exercise after changing the limits of the I_2 -loop as follows: $I_1 + 16 \leq I_2 \leq I_1 + 80$.
8. Find the new loop limits when the index variables of the double loop

```
do  $I_1 = 5, 100$ 
  do  $I_2 = 5, 100$ 
     $H(I_1, I_2)$ 
  enddo
enddo
```

are transformed by the matrix $\mathbf{U} = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix}$.

9. Repeat the previous exercise after changing the limits of the I_2 -loop as follows: $5 \leq I_2 \leq I_1 + 100$.

Part II

Data Dependence

Chapter 4

Basic Concepts

4.1 Introduction

Suppose that a sequential program has to be executed on a given machine. The program specifies a set of ‘actions’ that must be performed in a certain sequence, one after another. This is a total execution order. If necessary, a restructuring compiler for the given machine would try to find a new execution order (total or partial) for those actions, with the goal of utilizing the architectural features of the machine to the fullest extent. Utilization of the features usually means grouping the actions so that the actions in a group can be executed simultaneously, two groups can be executed independently or in an overlapped fashion, or a combination of these execution schemes can take place.

Any scrambling or grouping of the original sequence of actions is permissible as long as the meaning of the program does not change. To keep the meaning intact, we must obey the constraints of the underlying ‘dependence structure’ of the program. This dependence structure is determined by the way memory locations are referenced (read or written) and the way control flows, during the (sequential) execution of the program. In this volume, we are focusing on loop nests consisting of assignment statements only, so that the control-flow is quite straightforward. For now, dependence will mean only data dependence. Dependence between two statements is caused by

the intersection of the sets of memory locations referenced by the statements, and the ordering of those references during program execution. This relation can also be defined at other levels. For example, we will talk about dependence between statement instances and between iterations of loops.

In this chapter, we introduce the program model to be used in the rest of the book, and explain the fundamental dependence concepts. More specialized definitions will be presented as they are needed. The actual problem of computing the dependence relation by analyzing array subscripts is taken up in Chapter 5.

4.2 Sequential Loop Nest

We assume that the reader is familiar with computer programs in general and Fortran programs in particular. We are going to define most of the terms that will be used, but some programming experience is essential for a clear understanding of the concepts.

By a sequential (**do**) loop with stride 1, we mean a construct of the form:

```
L :  do I = p, q
      H(I)
      enddo
```

where p and q are integers, invariant for the loop. For L , the *index variable* is I , the *lower limit* is p , and the *upper limit* is q . If $p \leq q$, the index variable takes the $(q - p + 1)$ values $p, p + 1, \dots, q$, which are called the *index values* or *index points* of L . The loop is empty if $p > q$. The *body* of the loop L is $H(I)$; it is sometimes denoted simply by H .¹ For each index value i , we get an instance of the body, denoted by $H(i)$, which is an *iteration* of the loop. The $(q - p + 1)$ iterations $H(p), H(p + 1), \dots, H(q)$ are to be executed one after another in this order. There may be loops in the body of L , and L itself may be in the body of some other loop.

¹We use the notation $H(I)$ to emphasize that the body is a function of the index variable I .

Our object of study in this book is a perfect nest of sequential loops of the above type:

```

 $L_1 : \quad \text{do } I_1 = p_1, q_1$ 
 $L_2 : \quad \text{do } I_2 = p_2, q_2$ 
 $\vdots \quad \vdots$ 
 $L_m : \quad \text{do } I_m = p_m, q_m$ 
 $\quad H(I_1, I_2, \dots, I_m)$ 
 $\quad \text{enddo}$ 
 $\vdots$ 
 $\quad \text{enddo}$ 
 $\quad \text{enddo}$ 

```

where p_1, q_1 are integer constants; p_r, q_r are integer-valued functions² of I_1, I_2, \dots, I_{r-1} for $1 < r \leq m$; and $H(I_1, I_2, \dots, I_m)$ is a totally ordered set of assignment statements. It is a *perfect* loop nest in the sense that there are no statements between loops. We will write $\mathbf{L} = (L_1, L_2, \dots, L_m)$, and sometimes refer to this program as the loop nest \mathbf{L} or the loop nest (L_1, L_2, \dots, L_m) . When m has the value 1, 2, or 3, the perfect loop nest is a *single loop*, a *double loop*, or a *triple loop*, respectively.

For $1 < r \leq m$, the number of index values of L_r is a function of I_1, I_2, \dots, I_{r-1} . If $p_1 > q_1$, then the outermost loop L_1 is empty, and hence so is the entire nest \mathbf{L} . Suppose that $p_1 \leq q_1$, and choose a value i_1 of I_1 in $p_1 \leq i_1 \leq q_1$. If $p_2(i_1) > q_2(i_1)$, then the instance of the loop L_2 corresponding to the index value i_1 of L_1 is empty, so that the corresponding instance of the nest (L_2, L_3, \dots, L_m) is empty. In general, if for given values i_1, i_2, \dots, i_{r-1} of I_1, I_2, \dots, I_{r-1} we have

$$p_r(i_1, i_2, \dots, i_{r-1}) > q_r(i_1, i_2, \dots, i_{r-1}),$$

then the corresponding instance of the nest $(L_r, L_{r+1}, \dots, L_m)$ is empty.

We write $\mathbf{I} = (I_1, I_2, \dots, I_m)$ and call \mathbf{I} the *index vector* of the loop nest \mathbf{L} . The *index values* or *index points* of \mathbf{L} are the values of

²This, of course, includes the case where p_r or q_r are integer constants.

I, that is, the integer m -vectors (i_1, i_2, \dots, i_m) where

$$\left. \begin{array}{rcl} p_1 & \leq & i_1 & \leq & q_1 \\ p_2(i_1) & \leq & i_2 & \leq & q_2(i_1) \\ & \vdots & & & \\ p_m(i_1, i_2, \dots, i_{m-1}) & \leq & i_m & \leq & q_m(i_1, i_2, \dots, i_{m-1}) \end{array} \right\}$$

The *index space* of **L** is the subset of \mathbf{Z}^m consisting of all the index points. If the loop limits are independent of I_1, I_2, \dots, I_m , and $p_r \leq q_r$ for each r , then the number of index points is $\prod_{r=1}^m (q_r - p_r + 1)$.

The *body* of the loop nest **L** is $H(I_1, I_2, \dots, I_m)$ or $H(\mathbf{I})$. A given index value $\mathbf{i} = (i_1, i_2, \dots, i_m)$ defines a particular instance $H(\mathbf{i}) = H(i_1, i_2, \dots, i_m)$ of H , which is an *iteration* of **L**. By definition, the iterations of the loop nest are executed sequentially in the lexicographic order of the index values, that is, an iteration $H(\mathbf{i})$ is executed before an iteration $H(\mathbf{j})$ iff $\mathbf{i} \prec \mathbf{j}$.

A typical assignment statement in H is denoted by S , $S(\mathbf{I})$, or $S(I_1, I_2, \dots, I_m)$, and the instance of $S(\mathbf{I})$ for an index value \mathbf{i} is denoted by $S(\mathbf{i})$. Let \leq denote the total order of elements in H ; it is the lexical order of the statements. During the execution of an iteration $H(\mathbf{i})$ of the program, the instances of the statements in that iteration are executed in lexical order (by definition). If an iteration $H(\mathbf{i})$ is executed before an iteration $H(\mathbf{j})$, then obviously every statement instance in $H(\mathbf{i})$ is executed before every statement instance in $H(\mathbf{j})$. We collect these simple facts in the form of a lemma for future reference:

Lemma 4.1 *Let S and T denote two assignment statements in the body of loop nest **L**. In the execution of **L**, an instance $S(\mathbf{i})$ of S is executed before an instance $T(\mathbf{j})$ of T iff one of the following conditions holds:*

- (a) $\mathbf{i} \prec \mathbf{j}$
- (b) $\mathbf{i} = \mathbf{j}$ and $S < T$.

The index variables are always explicitly referred to as ‘index variables.’ When we talk about ‘variables’ in the program, we exclude I_1, I_2, \dots, I_m . Each variable in the program is assumed to be

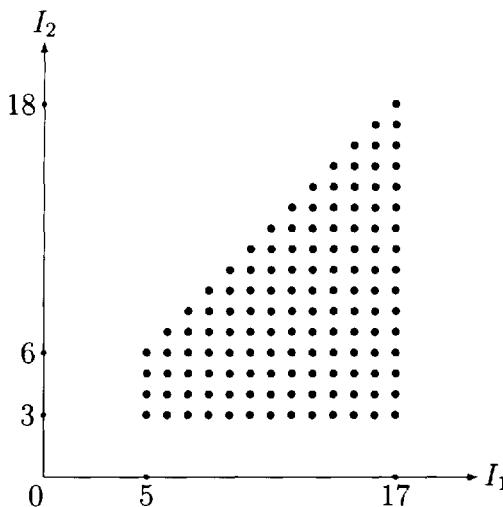


Figure 4.1: Index space of the loop nest in Example 4.1.

a scalar or a (scalar) element of an array. An assignment statement S in H has the form

$$S : \quad x = E$$

where x is a variable and E is an expression. For such a statement S , the *output variable* is x , and an *input variable* is any variable in E . We write $\text{OUT}(S) = \{x\}$ and denote by $\text{IN}(S)$ the set of all input variables of S . An output variable represents a memory write operation (a ‘store’), and an input variable represents a memory read operation (a ‘load’). The set of all output variables in H is denoted by $\text{OUT}(H)$, and the set of all input variables by $\text{IN}(H)$. Note that two or more occurrences of the same input variable in E are not counted as different variables.

Example 4.1 Consider the loop nest **L**:

```

L1 :   do I1 = 5,17
L2 :       do I2 = 3,I1 + 1
              H(I1,I2)
          enddo
      enddo
  
```

The outer loop has $(17 - 5 + 1)$ or 13 index values: $5, 6, \dots, 17$. For a given index value i_1 of L_1 , the inner loop has $(i_1 + 1 - 3 + 1)$ or $(i_1 - 1)$ index values: $3, 4, \dots, i_1 + 1$. The index space of the double loop is shown in Figure 4.1; it has $\sum_{i_1=5}^{17} (i_1 - 1)$ or 130 index points. The iterations of the loop nest are executed by taking their corresponding index points in lexicographic order. That is, we take the columns from left to right, and in each column take the points from bottom up.

Suppose that

$$S(I_1, I_2) : \quad A(I_1, I_2) = A(I_1, I_2) + C(I_1 + I_2) * A(I_1, I_2) + 1$$

is an assignment statement in H . Then the output and input variables of S are given by

$$\begin{aligned}\text{OUT}(S) &= \{A(I_1, I_2)\} \\ \text{IN}(S) &= \{A(I_1, I_2), C(I_1 + I_2)\}.\end{aligned}$$

To find the output and input variables of a particular instance of S , we substitute the corresponding value of (I_1, I_2) . For example, for the instance $S(8, 7)$ corresponding to $I_1 = 8, I_2 = 7$, we have

$$\begin{aligned}\text{OUT}(S(8, 7)) &= \{A(8, 7)\} \\ \text{IN}(S(8, 7)) &= \{A(8, 7), C(15)\}.\end{aligned}$$

EXERCISES 4.2

1. Draw the index space of the double loop

```

 $L_1 :$    do  $I_1 = p_1, q_1$ 
 $L_2 :$    do  $I_2 = p_2, q_2$ 
           $S(I_1, I_2)$ 
           $T(I_1, I_2)$ 
      enddo
  enddo

```

where

- (a) $p_1 = 12, q_1 = 31, p_2 = -4, q_2 = 14$
- (b) $p_1 = 0, q_1 = 20, p_2 = I_1 + 1, q_2 = I_1 + 16$

- (c) $p_1 = 5, q_1 = 30, p_2 = I_1 + 1, q_2 = 42$
- (d) $p_1 = 1, q_1 = 20, p_2 = I_1, q_2 = 2I_1 + 1$
- (e) $p_1 = 0, q_1 = 10, p_2 = 0, q_2 = \min(5, I_1)$
- (f) $p_1 = 0, q_1 = 5, p_2 = \max(0, I_1 - 3), q_2 = \min(2, I_1)$.

In each case, compute the number of index points and describe the execution order of the statement instances.

2. Find the number of index points for a triple loop when the loop limits are as follows:

- (a) $p_1 = 0, q_1 = 10, p_2 = 5, q_2 = 25, p_3 = 0, q_3 = I_2$
- (b) $p_1 = 0, q_1 = 10, p_2 = 1, q_2 = I_1, p_3 = 1, q_3 = I_2$.

4.3 Dependence Definitions

We will define several concepts of dependence, the two major ones being a relation of dependence between the statements of \mathbf{L} (the elements of H) and one between the iterations of \mathbf{L} (the instances of H). The relation of *dependence* between statements is denoted by δ , and is defined as follows: For two statements S and T , we have $S \delta T$ if there is an instance $S(\mathbf{i})$ of S , an instance $T(\mathbf{j})$ of T , and a memory location \mathcal{M} , such that

1. Both $S(\mathbf{i})$ and $T(\mathbf{j})$ reference (read or write) \mathcal{M} ;
2. $S(\mathbf{i})$ is to be executed before $T(\mathbf{j})$ in \mathbf{L} ;
3. During the execution of \mathbf{L} , the location \mathcal{M} is not written in the time period from the end of execution of $S(\mathbf{i})$ to the beginning of the execution of $T(\mathbf{j})$.

The statements S and T need not be distinct, but condition 2 requires that the instances $S(\mathbf{i})$ and $T(\mathbf{j})$ be distinct. The notation $S \delta T$ is read as ‘ T depends on S .’ The *dependence* of T on S is the set of all pairs $(S(\mathbf{i}), T(\mathbf{j}))$ that satisfy the above definition. Thus, we have $S \delta T$ iff the dependence of T on S is nonempty. The graph of the relation δ is called the *statement dependence graph* of \mathbf{L} .

Suppose that a statement T depends on a statement S . Let $S(\mathbf{i})$ and $T(\mathbf{j})$ denote a pair of instances, such that they (together with

some memory location \mathcal{M}) satisfy the above conditions. (There is at least one such pair.) We say that the instance $T(\mathbf{j})$ *depends* on the instance $S(\mathbf{i})$. Since $S(\mathbf{i})$ must be executed before $T(\mathbf{j})$, we have $\mathbf{i} \preceq \mathbf{j}$ (Lemma 4.1). If $\mathbf{i} \prec \mathbf{j}$, then the iteration $H(\mathbf{j})$ *depends* on the iteration $H(\mathbf{i})$. The graph of the relation of dependence between iterations is called the *iteration dependence graph*.³

Let $\mathbf{d} = \mathbf{j} - \mathbf{i}$, $\boldsymbol{\sigma} = \text{sig}(\mathbf{d})$, and $\ell = \text{lev}(\mathbf{d})$. (See Section 1.3.) Then, T depends on S with a *distance vector* \mathbf{d} , with a *direction vector* $\boldsymbol{\sigma}$, and at a *level* ℓ . It is sometimes convenient to say that \mathbf{d} is a distance (vector), $\boldsymbol{\sigma}$ a direction vector, and ℓ a level of the dependence of T on S .

Since $\mathbf{i} \preceq \mathbf{j}$ in the above definition, a distance vector or a direction vector of a dependence must always be (lexicographically) non-negative. There are $m+1$ possible values of the level: $1, 2, \dots, m+1$. If T depends on S at a level ℓ , $1 \leq \ell \leq m$, we say that the dependence of T on S is *carried* by the loop L_ℓ . The dependence of T on S is *loop-independent* if it is not carried by any loop (i.e., $m+1$ is the only level). Only loop-carried dependences between statements contribute towards dependence between iterations.

The transitive closure $\bar{\delta}$ of the relation δ is the relation of *indirect dependence*. Thus, a statement T is *indirectly dependent* on a statement S if there is a path from S to T in the statement dependence graph. In other words, we have $S \bar{\delta} T$ if there is a sequence of statements S_1, S_2, \dots, S_N such that

$$S = S_1, S_1 \delta S_2, \dots, S_{N-1} \delta S_N, S_N = T.$$

Similarly, an iteration $H(\mathbf{j})$ is *indirectly dependent* on an iteration $H(\mathbf{i})$ if there is a path from $H(\mathbf{i})$ to $H(\mathbf{j})$ in the iteration dependence graph.

Since a memory reference is either a ‘read’ or a ‘write,’ a pair of statement instances can reference the same memory location in four different ways. If there are two instances $S(\mathbf{i})$ and $T(\mathbf{j})$ satisfying the above definition such that $S(\mathbf{i})$ writes \mathcal{M} and $T(\mathbf{j})$ reads it, then

³Note that a statement instance cannot depend on itself, and an iteration cannot depend on itself. However, a statement may depend on itself when it represents more than one instance.

T is *flow dependent* on S . In this case, the value computed by $S(\mathbf{i})$ is to be actually used by $T(\mathbf{j})$. If $S(\mathbf{i})$ reads \mathcal{M} and then $T(\mathbf{j})$ writes it, then T is *anti-dependent* on S . Here, $S(\mathbf{i})$ must use the value in \mathcal{M} before it is changed by $T(\mathbf{j})$. Statement T is *output dependent* on S , if $S(\mathbf{i})$ and $T(\mathbf{j})$ both write \mathcal{M} . (The value computed by $T(\mathbf{j})$ is to be stored after the value computed by $S(\mathbf{i})$.) Finally, T is *input dependent* on S , if both $S(\mathbf{i})$ and $T(\mathbf{j})$ read \mathcal{M} . (The ‘read’ by $T(\mathbf{j})$ comes after the ‘read’ by $S(\mathbf{i})$.)

In the previous paragraph, we defined four new relations between statements. Let flow dependence, anti-dependence, output dependence, and input dependence be denoted by δ^f , δ^a , δ^o , and δ^i , respectively. (Thus, $S \delta^f T$ means T is flow dependent on S , etc.) These relations may overlap, and they constitute the main relation of dependence:

$$\delta = \delta^f \cup \delta^a \cup \delta^o \cup \delta^i.$$

The dependence between statements can also be classified in terms of the variables involved. The instances of the output variable of a statement S determine the memory locations written by the instances of S . On the other hand, the instances of the input variables of S determine the memory locations read by the instances of S . Let u denote a variable of a statement S and v a variable of a statement T . We say that the pair (u, v) *causes* a dependence of T on S , if the conditions in the definition of dependence hold, and \mathcal{M} is the memory location represented by both the instance of u for the index value \mathbf{i} , and the instance of v for the index value \mathbf{j} .

If u is the output variable of S and v an input variable of T , then (u, v) can cause only a flow dependence of T on S . If u is an input variable of S and v the output variable of T , then (u, v) can cause only an anti-dependence of T on S . The output variables of S and T can cause only an output dependence of T on S , and two input variables of the statements can cause only an input dependence.

The *flow dependence* of T on S is the set of all pairs $(S(\mathbf{i}), T(\mathbf{j}))$ that satisfy the definition of dependence such that $S(\mathbf{i})$ writes the location \mathcal{M} and $T(\mathbf{j})$ reads it. We can break up this set into (possibly overlapping) subsets that isolate the contributions of different pairs of variables of S and T . Let u denote the output variable of

S and v an input variable of T . The *flow dependence* of T on S caused by the variables (u, v) is the set of all pairs $(S(\mathbf{i}), T(\mathbf{j}))$ that satisfy the definition of dependence, such that the memory location \mathcal{M} is represented by the instance of u for the index value \mathbf{i} and also by the instance of v for the index value \mathbf{j} . These individual flow dependences make up the total flow dependence of T on S . Similar definitions can be given for the other three types (anti-, output, input) of dependence; we omit the details. When needed, we can associate a distance/direction vector or a level of the dependence of T on S with a particular type of dependence, or even with a particular pair of variables.

Example 4.2 Consider the single loop

```
L :      do I = 4, 200
         S :      A(I) = B(I) + C(I)
         T :      B(I + 2) = A(I - 1) + A(I - 3) + C(I - 1)
         U :      A(I + 1) = B(2I + 3) + 1
      enddo
```

The 197 iterations of L are executed in increasing order of the index values: $4, 5, \dots, 200$. The subscripts of the array elements are quite simple; we can easily figure out the details of the dependence relation by studying a few iterations. The first four iterations, corresponding to the index values 4, 5, 6 and 7, are shown below:

```
S(4) :      A(4) = B(4) + C(4)
T(4) :      B(6) = A(3) + A(1) + C(3)
U(4) :      A(5) = B(11) + 1
```

```
S(5) :      A(5) = B(5) + C(5)
T(5) :      B(7) = A(4) + A(2) + C(4)
U(5) :      A(6) = B(13) + 1
```

```
S(6) :      A(6) = B(6) + C(6)
T(6) :      B(8) = A(5) + A(3) + C(5)
U(6) :      A(7) = B(15) + 1
```

```
S(7) :      A(7) = B(7) + C(7)
T(7) :      B(9) = A(6) + A(4) + C(6)
U(7) :      A(8) = B(17) + 1
```

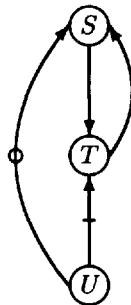


Figure 4.2: Statement dependence graph for Example 4.2.

Statement T is flow dependent on statement S . The flow dependence of T on S caused by the output variable $A(I)$ of S and the input variable $A(I - 1)$ of T is the set

$$\{(S(4), T(5)), (S(5), T(6)), (S(6), T(7)), \dots, (S(199), T(200))\}.$$

The flow dependence of T on S caused by the output variable $A(I)$ of S and the input variable $A(I - 3)$ of T is the set

$$\{(S(4), T(7)), (S(5), T(8)), (S(6), T(9)), \dots, (S(197), T(200))\}.$$

The union of these two sets gives the flow dependence of T on S . (The variables $A(I)$ and $C(I - 1)$ cannot cause a dependence, since the memory locations represented by the elements of the two arrays do not overlap.) The distance (vectors) for the flow dependence of T on S are 1 and 3, where 1 is the distance for the flow dependence caused by $(A(I), A(I - 1))$, and 3 is the distance for the flow dependence caused by $(A(I), A(I - 3))$.⁴

Statement S is also flow dependent on statement T . This flow dependence is caused by the output variable $B(I + 2)$ of T and the input variable $B(I)$ of S , and is given by

$$\{(T(4), S(6)), (T(5), S(7)), (T(6), S(8)), \dots, (T(198), S(200))\}.$$

The distance (vector) for the flow dependence of S on T is 2.

⁴Strictly speaking, we should write (1) and (3) for the distance vectors.

Statement S is output dependent on statement U . The output dependence of S on U is the set

$$\{(U(4), S(5)), (U(5), S(6)), (U(6), S(7)), \dots, (U(199), S(200))\}.$$

It has a distance of 1.

Statement T is anti-dependent on statement U . We find that the anti-dependence of T on U is the set

$$\{(U(4), T(9)), (U(5), T(11)), (U(6), T(13)), \dots, (U(99), T(199))\}.$$

This dependence has the distances: 5, 6, 7, ..., 100.

Note that statement T is not flow dependent on statement U . Whenever we have two instances $U(i)$ and $T(j)$, such that $U(i)$ is to be executed before $T(j)$ and they reference the same memory location, an instance of S writes the same location. For example, $U(4)$ and $T(6)$ reference the location represented by $A(5)$, but $S(5)$ writes this location and $S(5)$ comes between $U(4)$ and $T(6)$ in the execution of the program.

There is an input dependence of statement T on statement S , caused by the variables $C(I)$ and $C(I - 1)$. We will ignore input dependence for the most part in this book; it is not important for the loop transformations considered here.

The statement dependence graph of L is shown in Figure 4.2. (For quick recognition, we sometimes cross an anti-dependence edge and put a small circle on an output dependence edge.)

Example 4.3 Consider the double loop (L_1, L_2):

```

 $L_1 :$       do  $I_1 = 0, 4$ 
 $L_2 :$           do  $I_2 = 0, 4$ 
 $S :$            $A(I_1 + 1, I_2) = B(I_1, I_2) + C(I_1, I_2)$ 
 $T :$            $B(I_1, I_2 + 1) = A(I_1, I_2 + 1) + 1$ 
 $U :$            $D(I_1, I_2) = B(I_1, I_2 + 1) - 2$ 
                  enddo
                  enddo

```

The index space of (L_1, L_2) is shown in Figure 4.3. When executing the program, we process the 25 index points in the lexicographic

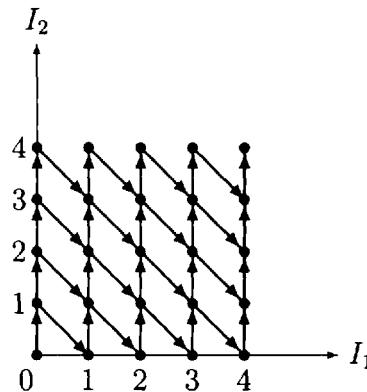


Figure 4.3: Iteration dependence graph for Example 4.3.

order; that is, the five columns are taken from left to right, and the points on each column are taken from bottom up.

Some of the iterations of (L_1, L_2) are shown below in the order in which they are to be executed:

$$S(0, 0) : \quad A(1, 0) = B(0, 0) + C(0, 0)$$

$$T(0, 0) : \quad B(0, 1) = A(0, 1) + 1$$

$$U(0, 0) : \quad D(0, 0) = B(0, 1) - 2$$

$$S(0, 1) : \quad A(1, 1) = B(0, 1) + C(0, 1)$$

$$T(0, 1) : \quad B(0, 2) = A(0, 2) + 1$$

$$U(0, 1) : \quad D(0, 1) = B(0, 2) - 2$$

$$S(0, 2) : \quad A(1, 2) = B(0, 2) + C(0, 2)$$

$$T(0, 2) : \quad B(0, 3) = A(0, 3) + 1$$

$$U(0, 2) : \quad D(0, 2) = B(0, 3) - 2$$

 \vdots
 \vdots

$$S(1, 0) : \quad A(2, 0) = B(1, 0) + C(1, 0)$$

$$T(1, 0) : \quad B(1, 1) = A(1, 1) + 1$$

$$U(1, 0) : \quad D(1, 0) = B(1, 1) - 2$$

$$S(1, 1) : \quad A(2, 1) = B(1, 1) + C(1, 1)$$

$$T(1, 1) : \quad B(1, 2) = A(1, 2) + 1$$

$$U(1, 1) : \quad D(1, 1) = B(1, 2) - 2$$

$$S(1, 2) : \quad A(2, 2) = B(1, 2) + C(1, 2)$$

$$T(1, 2) : \quad B(1, 3) = A(1, 3) + 1$$

$$U(1, 2) : \quad D(1, 2) = B(1, 3) - 2$$

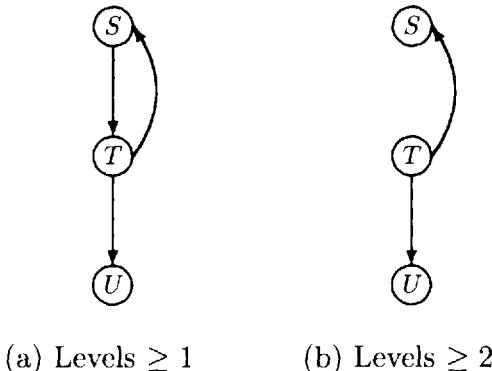


Figure 4.4: Statement dependence graphs for Example 4.3.

Statement T is flow dependent on statement S . The flow dependence of T on S is the set

$$\{(S(i_1, i_2), T(j_1, j_2)) : j_1 = i_1 + 1, j_2 = i_2 - 1, 0 \leq i_1 \leq 3, 1 \leq i_2 \leq 4\}.$$

It has one distance vector: $(1, -1)$, one direction vector: $(1, -1)$, and one level: 1. This dependence is therefore carried by loop L_1 .

Statement S is flow dependent on statement T . The flow dependence of S on T is the set

$$\{(T(i_1, i_2), S(j_1, j_2)) : j_1 = i_1, j_2 = i_2 + 1, 0 \leq i_1 \leq 4, 0 \leq i_2 \leq 3\}.$$

It has one distance vector: $(0, 1)$, one direction vector: $(0, 1)$, and one level: 2. This dependence is therefore carried by loop L_2 .

Statement U is flow dependent on statement T . The flow dependence of U on T is the set

$$\{(T(i_1, i_2), U(j_1, j_2)) : j_1 = i_1, j_2 = i_2, 0 \leq i_1 \leq 4, 0 \leq i_2 \leq 4\}.$$

It has one distance vector: $(0, 0)$, one direction vector: $(0, 0)$, and one level: 3. This dependence is therefore loop-independent.

The statement dependence graph of (L_1, L_2) is the first graph of Figure 4.4. The second graph in the same figure is the statement dependence graph of an instance of L_2 for a fixed value of I_1 .

Let $H(I_1, I_2)$ denote the body of (L_1, L_2) . An iteration $H(j_1, j_2)$ depends on an iteration $H(i_1, i_2)$ iff either $(j_1, j_2) = (i_1 + 1, i_2 - 1)$ or $(j_1, j_2) = (i_1, i_2 + 1)$. (Since the two iterations must be distinct, the dependence of U on T at level 3 does not contribute here.) The iteration dependence graph of (L_1, L_2) is shown in Figure 4.3, where we have identified an index point with the iteration it represents. Note, for example, that the iteration $H(3, 1)$ is indirectly dependent on the iteration $H(0, 0)$, but not on the iteration $H(1, 4)$.

EXERCISES 4.3

1. Prove that a sufficient condition for T to be indirectly dependent on S is that there exist index values \mathbf{i} and \mathbf{j} of the loop nest, such that
 - (a) $S(\mathbf{i})$ is executed before $T(\mathbf{j})$ in the execution of \mathbf{L} ;
 - (b) There is a memory location that is referenced by both instances.

Is this condition also necessary? Explain and give examples.

2. Why do we say that the dependence of T on S is *carried* by the loop L_ℓ if ℓ is a level of the dependence? Can the same dependence be carried by more than one loop? Give proof and/or examples.
3. Fill in the blanks:
 - (a) The dependence of T on S is carried by the loop L_ℓ , iff there is a distance vector \mathbf{d} of the dependence, such that ——.
 - (b) The dependence of T on S is carried by the loop L_ℓ , iff there is a direction vector σ of the dependence, such that ——.
 - (c) The dependence of T on S is loop-independent, iff each distance (direction) vector satisfies the condition ——.
4. Is it true that an iteration dependence graph is always acyclic?
5. For each single loop given below, find the dependences caused by all possible pairs of variables. Identify the type of each dependence and find its distances. Draw the statement dependence graph.

```
do  $I = 0, 100$ 
     $H(I)$ 
enddo
```

where $H(I)$ is the sequence

- (a) $A(I + 1) = A(I) + 1$
- $A(I) = A(I) + 2$

- (b) $A(I) = B(I + 2) + B(I) + B(I - 1) + B(I - 3)$
 $B(I) = A(I - 1) - 1$
- (c) $B(I) = A(I) + 3$
 $A(I - 1) = C(2I + 5) - 1$
 $A(I) = 2$
- (d) $A(2I) = B(I) + 1$
 $A(I) = C(I) + 2$
- (e) $A(I) = A(I) + B(I + 2)$
 $A(I - 1) = A(I - 1) + B(I + 1)$
- (f) $A(I) = C(I) + 1$
 $B(I) = A(I - 1) + A(2I - 5).$

6. For each double loop given below, find the dependences caused by all possible pairs of variables. Identify the type of each dependence and find its distance vectors, direction vectors, and levels. Draw the statement and iteration dependence graphs.

```
do  $I_1 = 3, 100$ 
  do  $I_2 = 4, 70$ 
     $H(I_1, I_2)$ 
  enddo
enddo
```

where the body H consists of

- (a) $A(I_1, I_2) = A(I_1 - 2, I_2 + 1) + 1$
- (b) $A(I_1 - 2, I_2 + 1) = A(I_1, I_2) + 1$
- (c) $A(I_1, I_2) = A(I_1, I_2 - 6) + 1$
- (d) $A(I_1, I_2) = A(I_1, I_2 + 6) + 1$
- (e) $A(I_1, I_2) = A(I_1, I_2 + 6) + A(I_1 - 4, I_2)$
- (f) $A(I_1, I_2) = A(I_2, I_1) + 1$
- (g) $A(I_1, I_2) = A(2I_1 + 1, I_2 + 3) + 1$
- (h) $A(I_1, I_2) = A(I_1, I_2) + 2$
- (i) $A(3I_1, I_2) = A(I_1, 2I_2) + 3$
- (j) $A(I_1, I_2) = C(I_1, I_2) - 1$
 $A(I_1 - 2, I_2 + 1) = B(I_1, I_2) + 1.$
- (k) $A(I_1 + 2, I_2) = B(2I_1, I_2) - 3$
 $B(2I_1, I_2 - 1) = A(I_1, I_2 + 2) + 12$
- (l) $A(I_1, I_2) = B(I_1 + 4, I_2 - 2) + B(I_1 + 2, I_2 - 3) + B(I_1, I_2 + 3)$
 $B(I_1, I_2) = C(I_1, I_2) + 12.$

Chapter 5

Linear Dependence Problem

5.1 Introduction

In Chapter 4, we introduced the basic dependence concepts in the context of a perfect nest \mathbf{L} of m loops. In this chapter, we consider the mathematical problem of actually computing the dependence of a statement on a statement, caused by a pair of variables. To derive the complete dependence information for \mathbf{L} , one needs to find the dependence caused by each pair of variables in the program.

We deal with variables that are elements of arrays; scalars can be easily included as a degenerate special case. The first part of the problem is to decide if the two sets of memory locations represented by instances of two variables in \mathbf{L} are disjoint. Since these sets would be necessarily disjoint if the variables were elements of two distinct arrays, we assume from the beginning that the variables being compared are elements of the same array. In real programs, an array subscript is usually a simple linear (affine) function of one or more index variables. We restrict ourselves to the case where each subscript of each variable is a linear function of the index variables of the loop nest. Comparison of two such variables that are elements of the same array leads to a system of linear diophantine equations. We also assume that the limits of a loop in \mathbf{L} are linear functions

of the index variables of loops that contain it. Those limits then will lead to a system of linear inequalities. In fact, more complex limits could be allowed as long as the range of each index variable is defined by a set of linear inequalities (e.g., the limits in the first transformed program in Example 2.1).

Let S and T denote two (not necessarily distinct) statements in the body of \mathbf{L} . Let u denote a variable of S and v a variable of T , and assume that they are elements of the same array. For ease of writing, explicit reference to these variables will often be suppressed. Thus, for example, we will say ‘ T does not depend on S ’ to mean that ‘the variable u of S and the variable v of T do not cause a dependence of T on S .’ The type (flow, anti-, output, input) of the dependence is irrelevant in this discussion. It is determined by whether u is an output or input variable of S , and whether v is an output or input variable of T . In a typical example, we will take u to be the output variable of S and v to be an input variable of T . Then, they may cause one or both of the following two dependences: a flow dependence of T on S , an anti-dependence of S on T . It is sometimes convenient to say that the variables u, v cause a dependence *between* S and T (or, there *is* a dependence *between* S and T), if they cause a dependence of T on S , or of S on T , or both.

Recall from Section 4.3 that the variables u and v cause a dependence of T on S , if there are index values \mathbf{i} and \mathbf{j} of \mathbf{L} such that

1. The instance of u for the index value \mathbf{i} and the instance of v for the index value \mathbf{j} represent the same memory location (denote it by \mathcal{M});
2. The instance $S(\mathbf{i})$ is executed before the instance $T(\mathbf{j})$, so that \mathcal{M} is referenced by $S(\mathbf{i})$ first;
3. During the execution of \mathbf{L} , the location \mathcal{M} is not written in the time period from the end of execution of $S(\mathbf{i})$ to the beginning of execution of $T(\mathbf{j})$.

From now on, we will not use the third condition. By not using condition 3, we may sometimes label as a dependence what is only

an indirect dependence (Exercise 4.3.1), but that will not affect any possible restructuring of \mathbf{L} . By definition, a valid loop transformation has to obey the constraints of dependence, and therefore it must also obey the constraints of indirect dependence (explain). In the absence of condition 3, the first condition will decide if there is a dependence between S and T , and given that the second will determine whether there is a dependence of T on S , or of S on T , or both.

The linear dependence problem and the general plan for its solution are described in sections 5.2–5.4. Two restricted versions of this problem (which cover many of the programs seen in practice) are discussed in sections 5.5 and 5.6. The following example will set the tone for this chapter. We will use it to informally introduce the matrix notation which will be defined more formally later on.

Example 5.1 Consider the problem of finding the dependence between S and T in the loop nest

```

 $L_1 :$       do  $I_1 = 1, 100$ 
 $L_2 :$           do  $I_2 = 1, I_1$ 
 $S :$              $X(2I_1 - 1, I_1 + I_2) = \dots$ 
 $T :$              $\dots = \dots X(3I_2, I_1 + 2) \dots$ 
                  enddo
                  enddo

```

caused by the two variables shown.

The output variable of S and the input variable of T are elements of a two-dimensional array X . The variable $X(2I_1 - 1, I_1 + I_2)$ can be written as $X(\mathbf{IA} + \mathbf{a}_0)$ where $\mathbf{I} = (I_1, I_2)$,

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{a}_0 = (-1, 0).$$

since

$$(2I_1 - 1, I_1 + I_2) = (I_1, I_2) \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} + (-1, 0).$$

Similarly, the variable $X(3I_2, I_1 + 2)$ can be written as $X(\mathbf{IB} + \mathbf{b}_0)$

where

$$\mathbf{B} = \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix} \text{ and } \mathbf{b}_0 = (0, 2).$$

The instance of $X(\mathbf{i}\mathbf{A} + \mathbf{a}_0)$ for an index value \mathbf{i} is $X(\mathbf{i}\mathbf{A} + \mathbf{a}_0)$, and the instance of $X(\mathbf{j}\mathbf{B} + \mathbf{b}_0)$ for an index value \mathbf{j} is $X(\mathbf{j}\mathbf{B} + \mathbf{b}_0)$. They will represent the same memory location iff $\mathbf{i}\mathbf{A} + \mathbf{a}_0 = \mathbf{j}\mathbf{B} + \mathbf{b}_0$ or

$$\mathbf{i}\mathbf{A} - \mathbf{j}\mathbf{B} = \mathbf{b}_0 - \mathbf{a}_0.$$

This is the dependence equation for the given problem. It can be written as

$$(\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = (1, 2)$$

where $(\mathbf{i}; \mathbf{j})$ is the result of adding the components of \mathbf{j} to \mathbf{i} , and $\begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix}$ is the result of adding the rows of $-\mathbf{B}$ to the rows of \mathbf{A} .

Writing out the vector and the matrix, we get the system:

$$(i_1, i_2, j_1, j_2) \begin{pmatrix} 2 & 1 \\ 0 & 1 \\ 0 & -1 \\ -3 & 0 \end{pmatrix} = (1, 2). \quad (5.1)$$

There are n equations in $2m$ variables, where n is the number of dimensions of the array X , and m is the number of loops in the loop nest; here $n = 2$ and $m = 2$.

To solve this system, we apply Theorem 3.6. By Algorithm 2.1, find two matrices

$$\mathbf{U} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 0 & 3 & 2 \\ 0 & 1 & 1 & 0 \end{pmatrix} \text{ and } \mathbf{S} = \begin{pmatrix} -1 & 1 \\ 0 & -1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

such that \mathbf{U} is unimodular, \mathbf{S} is echelon, and $\mathbf{U} \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{S}$. The system (5.1) has an (integer) solution in the variables i_1, i_2, j_1, j_2 iff the system (5.2) shown below has an (integer) solution in t_1, t_2, t_3, t_4 :

$$(t_1, t_2, t_3, t_4) \cdot \mathbf{S} = (1, 2). \quad (5.2)$$

If (5.2) has no solution, then two instances of the variables of S and T can never represent the same memory location, and hence they cannot cause a dependence between S and T . However, we find that (5.2) does have a solution. The general solution is given by:

$$t_1 = -1, t_2 = -3, t_3 \text{ and } t_4 \text{ are undetermined.}$$

Note that the number of integer parameters (t_1, t_2, t_3, t_4) whose values are determined by (5.2) is ρ where

$$\rho = \text{rank} \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \text{rank}(\mathbf{S}) = 2.$$

The number of undetermined integer parameters, which we denote by M , is given by $M = 2m - \rho = 2$.

The general solution to (5.1) can be written as

$$\begin{aligned} (i_1, i_2, j_1, j_2) &= (t_1, t_2, t_3, t_4) \cdot \mathbf{U} \\ &= (-1, -3, t_3, t_4) \cdot \mathbf{U} \\ &= (-1 + 3t_3, t_4, -3 + 3t_3 + t_4, -1 + 2t_3). \end{aligned} \quad (5.3)$$

Let \mathbf{U}_1 denote the matrix formed by the first two columns of \mathbf{U} and \mathbf{U}_2 the matrix formed by the last two columns. Then we have

$$\left. \begin{aligned} (i_1, i_2) &= (-1, -3, t_3, t_4) \cdot \mathbf{U}_1 \\ (j_1, j_2) &= (-1, -3, t_3, t_4) \cdot \mathbf{U}_2. \end{aligned} \right\} \quad (5.4)$$

Next, consider the constraints of the dependence problem. The index variables I_1 and I_2 must satisfy the inequalities:

$$\left. \begin{aligned} 1 &\leq I_1 && \leq 100 \\ 1 &\leq I_2 && \leq I_1 \end{aligned} \right\}$$

or

$$\left. \begin{aligned} 1 &\leq I_1 \\ 1 &\leq I_2 \\ I_1 &\leq 100 \\ -I_1 + I_2 &\leq 0. \end{aligned} \right\}$$

In matrix notation, we can write

$$\left. \begin{array}{l} \mathbf{p}_0 \leq \mathbf{IP} \\ \mathbf{IQ} \leq \mathbf{q}_0 \end{array} \right\}$$

where $\mathbf{p}_0 = (1, 1)$, $\mathbf{P} = \mathcal{I}_2$, $\mathbf{q}_0 = (100, 0)$, and $\mathbf{Q} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$.

Since (i_1, i_2) and (j_1, j_2) are values of $\mathbf{I} = (I_1, I_2)$, they must satisfy

$$\left. \begin{array}{l} \mathbf{p}_0 \leq (i_1, i_2)\mathbf{P} \\ (i_1, i_2)\mathbf{Q} \leq \mathbf{q}_0 \\ \mathbf{p}_0 \leq (j_1, j_2)\mathbf{P} \\ (j_1, j_2)\mathbf{Q} \leq \mathbf{q}_0. \end{array} \right\} \quad (5.5)$$

These are the dependence constraints for the given loop nest.

Substituting for (i_1, i_2) and (j_1, j_2) from (5.4) into (5.5), we get¹

$$\left. \begin{array}{l} \mathbf{p}_0 \leq (-1, -3, t_3, t_4) \cdot \mathbf{U}_1\mathbf{P} \\ (-1, -3, t_3, t_4) \cdot \mathbf{U}_1\mathbf{Q} \leq \mathbf{q}_0 \\ \mathbf{p}_0 \leq (-1, -3, t_3, t_4) \cdot \mathbf{U}_2\mathbf{P} \\ (-1, -3, t_3, t_4) \cdot \mathbf{U}_2\mathbf{Q} \leq \mathbf{q}_0. \end{array} \right\}$$

or (see Exercise 1)

$$\begin{aligned} & (t_3, t_4) \cdot \begin{pmatrix} -3 & 0 & -3 & -2 & 3 & -3 & 3 & -1 \\ 0 & -1 & -1 & 0 & 0 & 1 & 1 & -1 \end{pmatrix} \\ & \leq (-2, -1, -4, -2, 101, -1, 103, -2). \end{aligned} \quad (5.6)$$

For each (integer) solution (i_1, i_2, j_1, j_2) to (5.1) that satisfies (5.5), the corresponding integer vector (t_3, t_4) must satisfy (5.6). Conversely, for each (integer) solution (t_3, t_4) to (5.6), we get from (5.3) an (integer) solution (i_1, i_2, j_1, j_2) to (5.1) satisfying the constraints (5.5). If a solution to (5.6) exists, then that would indicate the existence of two iterations (i_1, i_2) and (j_1, j_2) of the loop nest,

¹We can get a more compact form of the inequalities in t_3, t_4 as follows: Rewrite (5.5) as $(\mathbf{p}_0; \mathbf{p}_0) \leq (i_1, i_2, j_1, j_2)\mathbf{P}$ and $(i_1, i_2, j_1, j_2)\mathbf{Q} \leq (\mathbf{q}_0; \mathbf{q}_0)$, and then use (5.3). The breakup of \mathbf{U} into \mathbf{U}_1 and \mathbf{U}_2 is needed anyway to get the dependence distance.

such that the instance $S(i_1, i_2)$ of S and the instance $T(j_1, j_2)$ of T reference the same memory location. If the iterations are such that $(i_1, i_2) \prec (j_1, j_2)$, then the instance $S(i_1, i_2)$ is executed before the instance $T(j_1, j_2)$, which indicates a dependence of T on S . Similarly, if the iterations are such that $(i_1, i_2) \succ (j_1, j_2)$, then S depends on T . If the iterations are identical, then there is dependence of T on S since $S < T$. (If S and T were the same statement, then $\mathbf{i} = \mathbf{j}$ would not imply any dependence.) If $(i_1, i_2) \prec (j_1, j_2)$, then

$$(j_1, j_2) - (i_1, i_2) = (-1, -3, t_3, t_4) \cdot (\mathbf{U}_2 - \mathbf{U}_1) = (-2 + t_4, -1 + 2t_3 - t_4)$$

is a distance vector for the dependence of T on S . If $(i_1, i_2) \succ (j_1, j_2)$, then

$$(i_1, i_2) - (j_1, j_2) = (-1, -3, t_3, t_4) \cdot (\mathbf{U}_1 - \mathbf{U}_2) = (2 - t_4, 1 - 2t_3 + t_4)$$

is a distance vector for the dependence of S on T .

EXERCISES 5.1

1. Show the details of the derivation of (5.6).
2. In Example 5.1, does statement T depend on statement S , or S on T ? Give reasons for your answer.

5.2 Dependence Equation

The model program **L** has the form:

```

 $L_1 :$     do  $I_1 = p_1, q_1$ 
 $L_2 :$     do  $I_2 = p_2, q_2$ 
 $\vdots$            $\vdots$ 
 $L_m :$     do  $I_m = p_m, q_m$ 
             $H(I_1, I_2, \dots, I_m)$ 
            enddo
 $\vdots$ 
enddo
enddo

```

where p_1 and q_1 are integer constants; p_r and q_r are integer-valued functions of I_1, I_2, \dots, I_{r-1} , for $1 < r \leq m$; and $H(I_1, I_2, \dots, I_m)$ is a sequence of assignment statements.

We assume that the variables in the program are array-elements with subscripts linear in I_1, I_2, \dots, I_m . A variable that is an element of an n -dimensional array X has the form

$$X(a_{11}I_1 + \dots + a_{m1}I_m + a_{10}, \dots, a_{1n}I_1 + \dots + a_{mn}I_m + a_{n0})$$

where the coefficients are all integer constants. In matrix notation, this variable can be written as $X(\mathbf{IA} + \mathbf{a}_0)$ where

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix},$$

$\mathbf{I} = (I_1, I_2, \dots, I_m)$ and $\mathbf{a}_0 = (a_{10}, a_{20}, \dots, a_{n0})$. The *coefficient matrix* of this element of X is \mathbf{A} , and its *constant term* is \mathbf{a}_0 .

Let S and T denote two (not necessarily distinct) statements in H . Let $X(\mathbf{IA} + \mathbf{a}_0)$ denote a variable of S and $X(\mathbf{IB} + \mathbf{b}_0)$ a variable of T , where X is an n -dimensional array, \mathbf{A} and \mathbf{B} are $m \times n$ integer matrices, and \mathbf{a}_0 and \mathbf{b}_0 are integer n -vectors. The instance of $X(\mathbf{IA} + \mathbf{a}_0)$ for an index value \mathbf{i} is $X(\mathbf{iA} + \mathbf{a}_0)$, and the instance of $X(\mathbf{IB} + \mathbf{b}_0)$ for an index value \mathbf{j} is $X(\mathbf{jB} + \mathbf{b}_0)$. They will represent the same memory location iff $\mathbf{iA} + \mathbf{a}_0 = \mathbf{jB} + \mathbf{b}_0$, that is, iff

$$\mathbf{iA} - \mathbf{jB} = \mathbf{b}_0 - \mathbf{a}_0. \quad (5.7)$$

The matrix equation (5.7) is the *dependence equation* for the variables $X(\mathbf{IA} + \mathbf{a}_0)$ and $X(\mathbf{IB} + \mathbf{b}_0)$. It consists of n scalar equations, and there are $2m$ variables: the m components of \mathbf{i} and the m components of \mathbf{j} . We can write (5.7) in the form

$$(\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b}_0 - \mathbf{a}_0 \quad (5.8)$$

where $(\mathbf{i}; \mathbf{j})$ is the $2m$ -vector obtained by adding the components of \mathbf{j} to the components of \mathbf{i} , and the coefficient matrix of the equation

is the $2m \times n$ matrix obtained by adding the rows of $-\mathbf{B}$ to the rows of \mathbf{A} .

The two variables $X(\mathbf{IA} + \mathbf{a}_0)$ and $X(\mathbf{IB} + \mathbf{b}_0)$ will cause a dependence between the statements S and T iff there are index points i and j satisfying the system of linear diophantine equations (5.8). This brings us to the basic test for dependence [Bane88a]:

Generalized GCD Test. *By Algorithm 2.1, find a $2m \times 2m$ unimodular matrix \mathbf{U} and a $2m \times n$ echelon matrix \mathbf{S} , such that*

$$\mathbf{U} \cdot \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{S}.$$

If there is no integer vector \mathbf{t} of size $2m$ satisfying

$$\mathbf{t}\mathbf{S} = \mathbf{b}_0 - \mathbf{a}_0, \quad (5.9)$$

then the variables $X(\mathbf{IA} + \mathbf{a}_0)$ of S and $X(\mathbf{IB} + \mathbf{b}_0)$ of T do not cause a dependence of T on S , nor of S on T .

The generalized GCD Test is a direct consequence of Theorem 3.6. By that theorem, the system of equations (5.8) has a solution iff there is an integer vector \mathbf{t} satisfying (5.9). The point of the test is that (5.9) is trivial to solve since \mathbf{S} is an echelon matrix.

Note that loop limits do not appear in the generalized GCD test. This test is a necessary condition for dependence, but not a sufficient one. If there is a solution \mathbf{t} to (5.9), then there must be a solution $(i; j) = \mathbf{t}\mathbf{U}$ to (5.8). That does not necessarily imply dependence, however, unless this i and this j are both index points of \mathbf{L} . The gcd test would be sufficient in the ideal case where the index space of \mathbf{L} is all of \mathbf{Z}^m . The constraints arising from the finiteness of the index space are discussed in the next section.

There is an alternate form of the generalized GCD test that uses Theorem 3.7 and Algorithm 2.3:

Generalized GCD Test (Alternate Form). *By Algorithm 2.3, find a $2m \times 2m$ unimodular matrix \mathbf{U} , an $n \times n$ unimodular matrix*

\mathbf{V} , and a $2m \times n$ diagonal matrix \mathbf{D} , such that

$$\mathbf{U} \cdot \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} \cdot \mathbf{V} = \mathbf{D}.$$

If there is no integer vector \mathbf{t} of size $2m$ satisfying

$$\mathbf{t}\mathbf{D} = (\mathbf{b}_0 - \mathbf{a}_0) \cdot \mathbf{V}, \quad (5.10)$$

then the variables $X(\mathbf{IA} + \mathbf{a}_0)$ of S and $X(\mathbf{IB} + \mathbf{b}_0)$ of T do not cause a dependence of T on S , nor of S on T .

The one-dimensional version of the Generalized GCD test is given below:

GCD Test. Let $X(a_1I_1 + a_2I_2 + \cdots + a_mI_m + a_0)$ denote a variable of a statement S and $X(b_1I_1 + b_2I_2 + \cdots + b_mI_m + b_0)$ a variable of a statement T , where X is a one-dimensional array. If $\gcd(a_1, a_2, \dots, a_m, b_1, b_2, \dots, b_m)$ does not divide $(b_0 - a_0)$, then the variables do not cause a dependence between S and T .

Here the dependence equation is

$$a_1i_1 + a_2i_2 + \cdots + a_mi_m - b_1j_1 - b_2j_2 - \cdots - b_mj_m = b_0 - a_0,$$

and the test follows from Theorem 3.5. The GCD test first appeared in [Coha73]; it has been described by many authors since then.

Example 5.2 To see if there is dependence between statements S and T in the double loop

```

 $L_1 :$       do  $I_1 = 1, 100$ 
 $L_2 :$           do  $I_2 = 12, 300$ 
 $S :$            $X(2I_1 + 4I_2 - 1) = \cdots$ 
 $T :$            $\cdots = \cdots X(4I_1 + 2I_2 + 6) \cdots$ 
          enddo
          enddo

```

we form the dependence equation:

$$2i_1 + 4i_2 - 1 = 4j_1 + 2j_2 + 6$$

or

$$2i_1 + 4i_2 - 4j_1 - 2j_2 = 7.$$

The gcd of the coefficients on the left hand side of the equation is 2. Since 2 does not divide 7, it follows from the GCD test that there is no dependence between S and T .

Example 5.3 In the program

```

L1 :      do I1 = 1,50
L2 :      do I2 = 12,I1 + 60
S :          X(2I1 + 2I2,I1 + 4I2) = ...
T :          ... = ... X(500 - 2I1 - 4I2,600 - I1 - 5I2) ...
        enddo
    enddo

```

we have

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 2 & 4 \end{pmatrix}, \mathbf{a}_0 = (0, 0), \mathbf{B} = \begin{pmatrix} -2 & -1 \\ -4 & -5 \end{pmatrix}, \mathbf{b}_0 = (500, 600).$$

The dependence equation (5.8) for the two variables of S and T is

$$(i_1, i_2, j_1, j_2) \begin{pmatrix} 2 & 1 \\ 2 & 4 \\ 2 & 1 \\ 4 & 5 \end{pmatrix} = (500, 600). \quad (5.11)$$

By Algorithm 2.1, we get the matrices

$$\mathbf{U} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & -1 \end{pmatrix} \text{ and } \mathbf{S} = \begin{pmatrix} 2 & 1 \\ 0 & 3 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

such that \mathbf{U} is unimodular, \mathbf{S} is echelon, and \mathbf{S} is the result of pre-multiplication of the coefficient matrix of (5.11) by \mathbf{U} . The general solution to the equation

$$(t_1, t_2, t_3, t_4)\mathbf{S} = (500, 600)$$

is $(250, 350/3, t_3, t_4)$ where t_3 and t_4 are undetermined. Since there is no integer solution, the Generalized GCD test implies that there is no dependence between statements S and T .

Note that the (one-dimensional) GCD Test applied separately to each of the two scalar equations in (5.11) will not lead to any definite conclusion. Those equations have separate (integer) solutions, but there is no simultaneous (integer) solution to the pair of equations.

EXERCISES 5.2

- Using one of the GCD tests, decide if there could be dependence between the statements S and T in the loop nest

```

do  $I_1 = p_1, q_1$ 
  do  $I_2 = p_2, q_2$ 
     $S :$        $u = \dots$ 
     $T :$        $\dots = \dots v \dots$ 
  enddo
enddo

```

where

- (a) $u = X(2I_1 + 3I_2 + 1), v = X(3I_1 - I_2 + 4)$
- (b) $u = X(4I_1 - 2I_2 - 1, -I_1 - 2I_2), v = X(I_1 + 2I_2 + 1, -2I_1 + 1)$
- (c) $u = X(2I_1 + 3I_2 - 2, 3I_1 + 4I_2 + 1), v = X(2I_1 + 3I_2 + 1, 3I_1 + 4I_2 + 3)$.

- Apply the alternate form of the generalized GCD test to the problem of Example 5.3.

5.3 Dependence Constraints

We assume that for $1 \leq r \leq m$, the limits p_r and q_r of the loop L_r in \mathbf{L} are given by

$$\begin{aligned} p_r(I_1, I_2, \dots, I_{r-1}) &= p_{r1}I_1 + p_{r2}I_2 + \dots + p_{r(r-1)}I_{r-1} + p_{r0} \\ q_r(I_1, I_2, \dots, I_{r-1}) &= q_{r1}I_1 + q_{r2}I_2 + \dots + q_{r(r-1)}I_{r-1} + q_{r0} \end{aligned}$$

where the coefficients are all integer constants. The index variables I_1, I_2, \dots, I_m satisfy the constraints:

$$p_r(I_1, I_2, \dots, I_{r-1}) \leq I_r \leq q_r(I_1, I_2, \dots, I_{r-1}). \quad (5.12)$$

Using the expressions for p_r given above, we can write the system of inequalities $p_r \leq I_r$ as

$$\left. \begin{array}{l} p_{10} \leq I_1 \\ p_{20} \leq -p_{21}I_1 + I_2 \\ p_{30} \leq -p_{31}I_1 - p_{32}I_2 + I_3 \\ \vdots \\ p_{m0} \leq -p_{m1}I_1 - p_{m2}I_2 - \cdots - p_{m(m-1)}I_{m-1} + I_m, \end{array} \right\}$$

or as $\mathbf{p}_0 \leq \mathbf{IP}$ where $\mathbf{p}_0 = (p_{10}, p_{20}, \dots, p_{m0})$ and

$$\mathbf{P} = \begin{pmatrix} 1 & -p_{21} & -p_{31} & \cdots & -p_{m1} \\ 0 & 1 & -p_{32} & \cdots & -p_{m2} \\ 0 & 0 & 1 & \cdots & -p_{m3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Similarly, the inequalities $I_r \leq q_r$ can be put in the form $\mathbf{IQ} \leq \mathbf{q}_0$ where $\mathbf{q}_0 = (q_{10}, q_{20}, \dots, q_{m0})$ and

$$\mathbf{Q} = \begin{pmatrix} 1 & -q_{21} & -q_{31} & \cdots & -q_{m1} \\ 0 & 1 & -q_{32} & \cdots & -q_{m2} \\ 0 & 0 & 1 & \cdots & -q_{m3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Summarizing, we can say that the polytope that is the index space of \mathbf{L} consists of all integer m -vectors $\mathbf{I} = (I_1, I_2, \dots, I_m)$ such that

$$\left. \begin{array}{l} \mathbf{p}_0 \leq \mathbf{IP} \\ \mathbf{IQ} \leq \mathbf{q}_0. \end{array} \right\} \quad (5.13)$$

The m -vectors \mathbf{p}_0 and \mathbf{q}_0 represent the constant parts of the loop limits, and the $m \times m$ upper triangular matrices \mathbf{P} and \mathbf{Q} represent the variable parts. Note that \mathbf{P} and \mathbf{Q} are unimodular matrices with $\det(\mathbf{P}) = \det(\mathbf{Q}) = 1$. For the loop nest \mathbf{L} , the *lower limit vector* is \mathbf{p}_0 , the *lower limit matrix* is \mathbf{P} , the *upper limit vector* is \mathbf{q}_0 , and the *upper limit matrix* is \mathbf{Q} .

We say that \mathbf{L} is a *rectangular* nest if the limits have no variable parts, that is, if $p_1, p_2, \dots, p_m, q_1, q_2, \dots, q_m$ are constants so that

$$\mathbf{p}_0 = (p_1, p_2, \dots, p_m), \mathbf{P} = \mathcal{I}_m, \mathbf{q}_0 = (q_1, q_2, \dots, q_m), \mathbf{Q} = \mathcal{I}_m.$$

The nest \mathbf{L} is *regular* if $\mathbf{P} = \mathbf{Q}$. Clearly, a rectangular nest is also a regular nest. The system of inequalities (5.13) describing the index space simplifies to

$$\mathbf{p}_0 \leq \mathbf{iP} \leq \mathbf{q}_0 \quad (5.14)$$

in the regular case, and to

$$\mathbf{p}_0 \leq \mathbf{i} \leq \mathbf{q}_0 \quad (5.15)$$

in the rectangular case.

Any solution $(\mathbf{i}; \mathbf{j})$ to the dependence equation ((5.7) or (5.8)) must be such that \mathbf{i} and \mathbf{j} are index points, that is, \mathbf{i} and \mathbf{j} satisfy (5.13):

$$\left. \begin{array}{lcl} \mathbf{p}_0 & \leq & \mathbf{iP} \\ & & \mathbf{iQ} \leq \mathbf{q}_0 \\ \mathbf{p}_0 & \leq & \mathbf{jP} \\ & & \mathbf{jQ} \leq \mathbf{q}_0. \end{array} \right\} \quad (5.16)$$

These inequalities are the *dependence constraints* for the loop nest \mathbf{L} . The *dependence problem* posed by the two variables $X(\mathbf{IA} + \mathbf{a}_0)$ and $X(\mathbf{IB} + \mathbf{b}_0)$ consists of solving the dependence equation (5.8) subject to the dependence constraints (5.16), and then properly classifying the solutions to derive the desired dependence information.

In examples 2.1 and 3.8, and in some of the exercises of Section 3.6, we noticed that a loop transformation usually changes a loop nest with fairly simple limits into one with rather complicated ones. We can easily extend our model of a loop nest to include such general limits. The idea is to accommodate loops where an index variable is restricted on either side by one or more linear functions with rational coefficients. In the extended model, we would take \mathbf{p}_0 and \mathbf{q}_0 to be rational m -vectors, and \mathbf{P} and \mathbf{Q} to be rational matrices with m rows and m or more columns. The dependence

constraints for the general model are still given by (5.16). The number of inequalities probably increases, but the inherent complexity of the problem does not change. We will consider this general situation in an example given below (and in some of the exercises), but after that will continue to assume that the forms of \mathbf{p}_0 , \mathbf{q}_0 , \mathbf{P} and \mathbf{Q} are as described in this section.

Example 5.4 The index space of the double loop

```

 $L_1 :$       do  $I_1 = 1, 50$ 
 $L_2 :$       do  $I_2 = 2I_1 - 1, 3I_1 + 4$ 
 $\vdots$ 
enddo
enddo
```

is defined by the system of inequalities:

$$\left. \begin{array}{l} 1 \leq I_1 \leq 50 \\ 2I_1 - 1 \leq I_2 \leq 3I_1 + 4 \end{array} \right\}$$

or

$$\left. \begin{array}{l} 1 \leq I_1 \\ -1 \leq -2I_1 + I_2 \\ I_1 \leq 50 \\ -3I_1 + I_2 \leq 4. \end{array} \right\}$$

We can write this system in the form:

$$\begin{aligned} (1, -1) &\leq (I_1, I_2) \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix} \\ (I_1, I_2) \begin{pmatrix} 1 & -3 \\ 0 & 1 \end{pmatrix} &\leq (50, 4). \end{aligned}$$

Thus, $\mathbf{p}_0 = (1, -1)$ is the lower limit vector and $\mathbf{q}_0 = (50, 4)$ is the upper limit vector for this loop nest. Also, the lower and upper limit matrices are given by

$$\mathbf{P} = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix} \text{ and } \mathbf{Q} = \begin{pmatrix} 1 & -3 \\ 0 & 1 \end{pmatrix}.$$

Example 5.5 The index space of the double loop

```

 $L_1 :$       do  $I_1 = 20, 400$ 
 $L_2 :$           do  $I_2 = \lceil \max(5, (I_1 - 100)/3) \rceil, \min(100, 2I_1 + 5)$ 
                  :
enddo
enddo

```

consists of all integer vectors (I_1, I_2) such that

$$\left. \begin{array}{l} 20 \leq I_1 \leq 400 \\ \max(5, (I_1 - 100)/3) \leq I_2 \leq \min(100, 2I_1 + 5), \end{array} \right\}$$

that is, all (I_1, I_2) such that

$$\left. \begin{array}{l} 20 \leq I_1 \\ 5 \leq I_2 \\ -\frac{100}{3} \leq -\frac{1}{3}I_1 + I_2 \end{array} \right\}$$

and

$$\left. \begin{array}{l} I_1 \leq 400 \\ I_2 \leq 100 \\ -2I_1 + I_2 \leq 5. \end{array} \right\}$$

The last two sets can be written as

$$\left. \begin{array}{l} \mathbf{p}_0 \leq \mathbf{iP} \\ \mathbf{p}_0 \leq \mathbf{jP} \\ \mathbf{p}_0 \leq \mathbf{iQ} \\ \mathbf{p}_0 \leq \mathbf{jQ} \\ \mathbf{q}_0 \leq \mathbf{q}_0 \end{array} \right\}$$

where

$$\mathbf{p}_0 = (20, 5, -\frac{100}{3}), \quad \mathbf{q}_0 = (400, 100, 5)$$

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & -\frac{1}{3} \\ 0 & 1 & 1 \end{pmatrix}, \quad \mathbf{Q} = \begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & 1 \end{pmatrix}.$$

EXERCISES 5.3

1. Find $\mathbf{p}_0, \mathbf{q}_0, \mathbf{P}, \mathbf{Q}$ for each of the following loop nests, and write down the dependence constraints:
 - (a) The loop nests of Exercise 4.2.1;
 - (b) The loop nests of Exercise 4.2.2;
 - (c) The original and transformed loop nests of Example 3.8.

5.4 Dependence Algorithm

We now return to the discussion of the dependence problem started in Section 5.2. We will assume—without any loss in generality—that $S \leq T$. The variable $X(\mathbf{IA} + \mathbf{a}_0)$ of statement S and the variable $X(\mathbf{IB} + \mathbf{b}_0)$ of statement T will cause a dependence between S and T , iff there are index points \mathbf{i} and \mathbf{j} of \mathbf{L} satisfying the dependence equation (5.8):

$$(\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b}_0 - \mathbf{a}_0.$$

Thus, we want to know if there is a solution $(\mathbf{i}; \mathbf{j})$ to this equation such that \mathbf{i} and \mathbf{j} satisfy the dependence constraints (5.16). Knowing all such solutions, we can compute the dependence of T on S , and of S on T . We can also compute the distance vectors, the direction vectors, and the levels of a dependence. A high-level algorithm for the solution of the linear dependence problem is given below:

Algorithm 5.1 (Linear Dependence Algorithm) Consider a perfect nest of loops $\mathbf{L} = (L_1, L_2, \dots, L_m)$ with index vector \mathbf{I} , lower limit vector \mathbf{p}_0 , lower limit matrix \mathbf{P} , upper limit vector \mathbf{q}_0 , and upper limit matrix \mathbf{Q} . Let S and T denote two statements in \mathbf{L} such that $S \leq T$. Let $X(\mathbf{IA} + \mathbf{a}_0)$ denote a variable of S and $X(\mathbf{IB} + \mathbf{b}_0)$ a variable of T , where X is an n -dimensional array, \mathbf{A} and \mathbf{B} are $m \times n$ integer matrices, and \mathbf{a}_0 and \mathbf{b}_0 are integer n -vectors.

Given the lexical ordering of S and T , the vectors $\mathbf{p}_0, \mathbf{q}_0, \mathbf{a}_0, \mathbf{b}_0$, and the matrices $\mathbf{P}, \mathbf{Q}, \mathbf{A}, \mathbf{B}$, this algorithm gives the major steps

of a process that decides if the two variables cause a dependence between S and T . In case they do cause a dependence, we also show how to compute the dependence of T on S , the dependence of S on T , and the distance vectors, direction vectors, and levels of each dependence.

1. By Algorithm 2.1, find a $2m \times 2m$ unimodular matrix \mathbf{U} and a $2m \times n$ echelon matrix \mathbf{S} , such that

$$\mathbf{U} \cdot \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{S}.$$

2. Solve the system of diophantine equations:

$$\mathbf{t}\mathbf{S} = \mathbf{b}_0 - \mathbf{a}_0 \quad (5.17)$$

and find the general solution \mathbf{t} if a solution exists.

[This system is easy to solve since \mathbf{S} is an echelon matrix.]

3. (Generalized GCD Test) If there is no (integer) solution to (5.17), then there is no dependence between statements S and T , and the algorithm is terminated.
4. Set

$$\begin{aligned} \rho &\leftarrow \text{rank}(\mathbf{S}) \\ M &\leftarrow 2m - \rho. \end{aligned}$$

[Note that ρ is the number of nonzero rows of \mathbf{S} . The first ρ components of \mathbf{t} are known integers, and the last M components are undetermined.]

5. Set \mathbf{U}_1 to the $2m \times m$ matrix consisting of the first m columns of \mathbf{U} , and \mathbf{U}_2 to the $2m \times m$ matrix consisting of the last m columns of \mathbf{U} .

[By Theorem 3.6, the general solution to the dependence equation is $(\mathbf{i}; \mathbf{j}) = \mathbf{t}\mathbf{U}$. Hence, $\mathbf{i} = \mathbf{t}\mathbf{U}_1$ and $\mathbf{j} = \mathbf{t}\mathbf{U}_2$.]

6. [Since i and j are index points, they must satisfy the dependence constraints (5.16). Substitute $i = tU_1$ and $j = tU_2$ in the constraints to get the set of inequalities in the undetermined components of t .]

If the system of $4m$ inequalities

$$\left. \begin{array}{l} p_0 \leq t(U_1 P) \\ \quad t(U_1 Q) \leq q_0 \\ p_0 \leq t(U_2 P) \\ \quad t(U_2 Q) \leq q_0 \end{array} \right\} \quad (5.18)$$

in M variables (the undetermined components of t) has no (integer) solution, then there is no dependence between the statements S and T , and the algorithm is terminated.

7. Select case based on the lexical ordering of S and T :

Case ($S < T$): Assign to the dependence of T on S the set

$$\{(S(i), T(j)) : i = tU_1, j = tU_2\}$$

where t is an arbitrary solution to (5.17) that satisfies (5.18) and $tU_1 \preceq tU_2$.

Assign to the dependence of S on T the set

$$\{(T(j), S(i)) : i = tU_1, j = tU_2\}$$

where t is an arbitrary solution to (5.17) that satisfies (5.18) and $tU_1 \succ tU_2$.

Case ($S = T$): Assign to the dependence of S on itself the set

$$\{(S(i), S(j)) : i = tU_1, j = tU_2\}$$

where t is an arbitrary solution to (5.17) that satisfies (5.18) and $tU_1 \prec tU_2$.

If the dependence of T on S is empty, then T does not depend on S . If the dependence of S on T is empty, then S does not depend on T .

8. For each instance $T(\mathbf{t}\mathbf{U}_2)$ of T depending on each instance $S(\mathbf{t}\mathbf{U}_1)$ of S , compute the corresponding distance vector (of the dependence of T on S): $\mathbf{t}(\mathbf{U}_2 - \mathbf{U}_1)$.

For each instance $S(\mathbf{t}\mathbf{U}_1)$ of S depending on each instance $T(\mathbf{t}\mathbf{U}_2)$ of T , compute the corresponding distance vector (of the dependence of S on T): $\mathbf{t}(\mathbf{U}_1 - \mathbf{U}_2)$.

For each distance vector \mathbf{d} , compute the corresponding direction vector $\text{sig}(\mathbf{d})$ and the corresponding level $\text{lev}(\mathbf{d})$.

9. Terminate the algorithm.

□

The hardest part of the above algorithm is to determine if the system of inequalities (5.18) has a solution. In general, we can use any known integer programming algorithm (e.g., Gomory's cutting plane method [Schr87]). However, it has been argued that such a general algorithm should not be included as part of a dependence test in the compiler, based on the following empirical facts:

1. In a typical sequential program, the compiler must test for dependence a large number of times.
2. A general integer programming method is usually time consuming.
3. The subscripts seen in real programs are usually very simple.

Much effort has been spent over the years on discovering simpler methods for dependence testing. A full discussion of those methods is beyond our current scope; we will only describe here a couple of special cases where the dependence problem can be solved rather easily. These cases cover much of what is seen in real programs, and they will provide us with meaningful examples for loop transformations.

Note that a system of linear inequalities is trivial to solve if the number of unknowns in it is 0 or 1. Thus, the system (5.18) becomes trivial if $M = 0$ or $M = 1$. These cases are illustrated in examples 5.6 and 5.7 given below.

Let $\text{rank}(\mathbf{S}) = 2m$, so that $M = 0$. In this case, equation (5.17) either has a unique solution or no solution at all. If (5.17) has a unique solution \mathbf{t} , then the dependence equation (5.8) has a unique solution $(i; j) = \mathbf{t}\mathbf{U}$, and the inequalities (5.18) do not involve any variables (so that they are easy to solve). Otherwise, the dependence equation has no solution. Typically, we have the situation $\text{rank}(\mathbf{S}) = 2m$ when the matrix \mathbf{S} is square ($n = 2m$, i.e., the number of dimensions of X is two times the number of loops in \mathbf{L}), and nonsingular. There are again two important special cases of a nonsingular \mathbf{S} : a unimodular matrix, and a square diagonal matrix with nonzero diagonal elements.

Let $\text{rank}(\mathbf{S}) = 2m - 1$, so that $M = 1$. If (5.17) has a solution, then the general solution \mathbf{t} contains one undetermined component. The general solution to the dependence equation has one arbitrary parameter, and there is only one variable in the system of inequalities (5.18). One can easily decide in this case if (5.18) has an integer solution.

When $M \geq 2$, it may still be possible to break up (5.18) into subsystems such that each subsystem has no more than one variable. We will show in the next two sections how certain restricted versions of the dependence problem can be solved this way.

Example 5.6 Consider the dependence problem posed by the two variables shown in the loop:

```
L :      do I = 10, 200
        S :      X(2I - 2, I + 3) = ...
        T :      ... = ... X(300 - I, 2I + 9) ...
    enddo
```

Here we have

$$\begin{aligned} m &= 1, & n &= 2 \\ \mathbf{P} &= (1), & \mathbf{p}_0 &= (10), & \mathbf{Q} &= (1), & \mathbf{q}_0 &= (200) \\ \mathbf{A} &= (2, 1), & \mathbf{a}_0 &= (-2, 3), & \mathbf{B} &= (-1, 2), & \mathbf{b}_0 &= (300, 9). \end{aligned}$$

The dependence equation (5.8) is

$$(i, j) \cdot \begin{pmatrix} 2 & 1 \\ 1 & -2 \end{pmatrix} = (302, 6).$$

The steps of Algorithm 5.1 would be as follows:

1. By Algorithm 2.1, find a unimodular matrix $\mathbf{U} = \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix}$ and an echelon matrix $\mathbf{S} = \begin{pmatrix} 1 & -2 \\ 0 & 5 \end{pmatrix}$ such that

$$\begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 1 & -2 \end{pmatrix} = \begin{pmatrix} 1 & -2 \\ 0 & 5 \end{pmatrix}.$$

2. Equation (5.17) is

$$(t_1, t_2) \cdot \begin{pmatrix} 1 & -2 \\ 0 & 5 \end{pmatrix} = (302, 6).$$

It has a unique solution: $(t_1, t_2) = (302, 122)$.

3. The Generalized GCD test is passed.

4. Set $\rho \leftarrow 2$ and $M \leftarrow 0$.

5. The two submatrices of \mathbf{U} are:

$$\mathbf{U}_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \text{and} \quad \mathbf{U}_2 = \begin{pmatrix} 1 \\ -2 \end{pmatrix}.$$

6. The system (5.18) is

$$\left. \begin{array}{l} 10 \leq (t_1, t_2) \begin{pmatrix} 0 \\ 1 \end{pmatrix} (1) \\ (t_1, t_2) \begin{pmatrix} 0 \\ 1 \end{pmatrix} (1) \leq 200 \\ 10 \leq (t_1, t_2) \begin{pmatrix} 1 \\ -2 \end{pmatrix} (1) \\ (t_1, t_2) \begin{pmatrix} 1 \\ -2 \end{pmatrix} (1) \leq 200. \end{array} \right\}$$

Setting $(t_1, t_2) = (302, 122)$, we get a system in zero variables:

$$\left. \begin{array}{rcl} 10 & \leq & 122 \\ & & 122 \leq 200 \\ 10 & \leq & 58 \\ & & 58 \leq 200 \end{array} \right\}$$

which is consistent.

7. For the unique value of (t_1, t_2) , we have $(t_1, t_2)\mathbf{U}_1 = 122$ and $(t_1, t_2)\mathbf{U}_2 = 58$. Since $122 > 58$, the dependence of T on S is empty and the dependence of S on T consists of the single instance pair $(T(58), S(122))$. Thus, T does not depend on S , but S depends on T .
8. The dependence of S on T has a distance (vector) 64, a direction (vector) 1, and a level 1.

Example 5.7 For the double loop

```

 $L_1 :$       do  $I_1 = 10, 200$ 
 $L_2 :$           do  $I_2 = 7, 167$ 
 $S :$              $X(2I_1 + 3, 5I_2 - 1, I_2) = \dots$ 
 $T :$              $\dots = \dots X(I_1 - 1, 2I_1 - 6, 3I_2 + 2) \dots$ 
            enddo
        enddo
    
```

we have

$$m = 2, \quad n = 3$$

$$\mathbf{P} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathbf{p}_0 = (10, 7)$$

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathbf{q}_0 = (200, 167)$$

$$\mathbf{A} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 5 & 1 \end{pmatrix}, \quad \mathbf{a}_0 = (3, -1, 0)$$

$$\mathbf{B} = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}, \quad \mathbf{b}_0 = (-1, -6, 2).$$

The dependence equation (5.8) is

$$(i_1, i_2, j_1, j_2) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 5 & 1 \\ -1 & -2 & 0 \\ 0 & 0 & -3 \end{pmatrix} = (-4, -5, 2).$$

The steps of Algorithm 5.1 would be as follows:

1. Find the matrices \mathbf{U} and \mathbf{S} :

$$\mathbf{U} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 2 & 0 \\ 5 & 4 & 10 & 1 \\ 15 & 12 & 30 & 4 \end{pmatrix} \quad \text{and} \quad \mathbf{S} = \begin{pmatrix} -1 & -2 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}.$$

2. Equation (5.17) is

$$(t_1, t_2, t_3, t_4) \cdot \begin{pmatrix} -1 & -2 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} = (-4, -5, 2). \quad (5.19)$$

It has the general solution:

$$(t_1, t_2, t_3, t_4) = (4, 3, -1, t_4)$$

where t_4 is arbitrary.

3. The Generalized GCD test is passed.
4. Set $\rho \leftarrow 3$ and $M \leftarrow 1$.
5. The two submatrices of \mathbf{U} are:

$$\mathbf{U}_1 = \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 5 & 4 \\ 15 & 12 \end{pmatrix} \quad \text{and} \quad \mathbf{U}_2 = \begin{pmatrix} 1 & 0 \\ 2 & 0 \\ 10 & 1 \\ 30 & 4 \end{pmatrix}.$$

6. The system of inequalities (5.18) in one variable is

$$\left. \begin{array}{l} 10 \leq -2 + 15t_4 \\ 7 \leq -1 + 12t_4 \\ -2 + 15t_4 \leq 200 \\ -1 + 12t_4 \leq 167 \\ 10 \leq 30t_4 \\ 7 \leq -1 + 4t_4 \\ 30t_4 \leq 200 \\ -1 + 4t_4 \leq 167. \end{array} \right\} \quad (5.20)$$

There are only 5 integer values of t_4 satisfying this system:
 $t_4 = 2, 3, 4, 5, 6$.

7. For a given solution $\mathbf{t} = (4, 3, -1, t_4)$ to (5.19), the corresponding index values \mathbf{i} and \mathbf{j} are

$$\begin{aligned} \mathbf{i} &= \mathbf{t}\mathbf{U}_1 = (-2 + 15t_4, -1 + 12t_4) \\ \mathbf{j} &= \mathbf{t}\mathbf{U}_2 = (30t_4, -1 + 4t_4). \end{aligned}$$

This \mathbf{t} also satisfies (5.20) iff $2 \leq t_4 \leq 6$. It turns out that for each value of t_4 in $2 \leq t_4 \leq 6$, we have $\mathbf{i} \prec \mathbf{j}$. Thus, the dependence of T on S is the set

$$\{(S(-2 + 15t_4, -1 + 12t_4), T(30t_4, -1 + 4t_4)) : 2 \leq t_4 \leq 6\}$$

which consists of the statement instance pairs

$$\begin{aligned} (S(28, 23), T(60, 7)), \quad (S(43, 35), T(90, 11)), \\ (S(58, 47), T(120, 15)), \quad (S(73, 59), T(150, 19)), \\ (S(88, 71), T(180, 23)). \end{aligned}$$

Also, the dependence of S on T is empty.

8. The dependence of T on S has the distance vectors:

$$\{(32, -16), (47, -24), (62, -32), (77, -40), (92, -48)\},$$

the direction vector $(1, -1)$, and the level 1.

EXERCISES 5.4

- Let \mathbf{d} denote an arbitrary distance vector of the dependence of T on S . Find the ranges of the components of \mathbf{d} when \mathbf{L} is a rectangular loop nest. Generalize the results to the case when \mathbf{L} is regular.
- In Example 5.3, change the constant term in the second subscript of the input variable of T from 600 to 601. Apply Algorithm 5.1; go up to the point where the inequalities in the undetermined components of \mathbf{t} are obtained.
- Apply Algorithm 5.1 to the loop nest

```

do  $I_1 = 1, 200$ 
    do  $I_2 = 1, 200$ 
         $S : \quad X(I_1 + 3, I_2 - 1, I_1 + I_2 + 1, I_1 + I_2 - 5) = \dots$ 
         $T : \quad \dots = \dots X(I_1 + I_2 + 1, I_1 + I_2 + 1, I_2 + 2, I_1 + I_2 - 7) \dots$ 
    enddo
enddo

```

- Repeat Example 5.7 after changing the loop limits as follows:
 - $0 \leq I_1 \leq 200, 0 \leq I_2 \leq 300$
 - $-100 \leq I_1 \leq 10, -10 \leq I_2 \leq 250$
 - $0 \leq I_1 \leq 200, I_1 \leq I_2 \leq I_1 + 150$
 - $0 \leq I_1 \leq 200, 0 \leq I_2 \leq I_1 + 150$.
- Customize Algorithm 5.1 so that it decides if the variables $X(\mathbf{IA} + \mathbf{a}_0)$ of S and $X(\mathbf{IB} + \mathbf{b}_0)$ of T cause a dependence of T on S with a given
 - direction vector $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_m)$
 - dependence level ℓ .

Take advantage of equations like $i_r = j_r$ when $\sigma_r = 0$ or $r < \ell$.

5.5 Regular Loop Nest

We assume in this section that \mathbf{L} is a regular loop nest, so that $\mathbf{P} = \mathbf{Q}$. The index space of \mathbf{L} consists of all integer m -vectors \mathbf{I} that satisfy (5.14):

$$\mathbf{p}_0 \leq \mathbf{IP} \leq \mathbf{q}_0.$$

The dependence constraints for this case are:

$$\left. \begin{array}{l} p_0 \leq iP \\ p_0 \leq jP \end{array} \right\} \leq q_0 \quad (5.21)$$

We consider a simplified version of the dependence problem where the coefficient matrices of the variables $X(\mathbf{IA} + \mathbf{a}_0)$ and $X(\mathbf{IB} + \mathbf{b}_0)$ are the same, that is, $\mathbf{A} = \mathbf{B}$. The dependence equation (5.7) then reduces to

$$(\mathbf{i} - \mathbf{j})\mathbf{A} = \mathbf{b}_0 - \mathbf{a}_0 \quad (5.22)$$

or to

$$\mathbf{d}\mathbf{A} = \mathbf{a}_0 - \mathbf{b}_0 \quad (5.23)$$

where $\mathbf{d} = \mathbf{j} - \mathbf{i}$. Writing $\mathbf{j} = \mathbf{i} + \mathbf{d}$ in (5.21) and then eliminating \mathbf{i} , we get the constraints on \mathbf{d} :

$$p_0 - q_0 \leq \mathbf{d}\mathbf{P} \leq q_0 - p_0. \quad (5.24)$$

If $(\mathbf{i}; \mathbf{j})$ is a solution to (5.22) that satisfies (5.21), then $\mathbf{d} = \mathbf{j} - \mathbf{i}$ is a solution to (5.23) that satisfies (5.24). Conversely, if \mathbf{d} is a solution to (5.23) satisfying (5.24), then there exists a vector \mathbf{i} , such that $(\mathbf{i}; \mathbf{j})$ with $\mathbf{j} = \mathbf{i} + \mathbf{d}$ is a solution to (5.22) that satisfies (5.21). (The proof is left to the reader. See Exercise 1.)

The system (5.23) consists of n scalar equations in m variables, while the general dependence equation (5.7) represents the same number of scalar equations in $2m$ variables. Also, there are $2m$ (one-sided) scalar inequalities in (5.24), as opposed to $4m$ such inequalities in (5.21). Thus, this special dependence problem is smaller in size than the general problem.

Suppose \mathbf{d} is a solution to (5.23) satisfying (5.24). If $\mathbf{d} \succ \mathbf{0}$, then T depends on S and \mathbf{d} is a distance of that dependence. This distance is such that the instance $T(\mathbf{i} + \mathbf{d})$ depends on the instance $S(\mathbf{i})$ whenever \mathbf{i} and $\mathbf{i} + \mathbf{d}$ are both index points of \mathbf{L} . A distance vector with this property is said to be *uniform*. Similarly, the existence of a (lexicographically) negative \mathbf{d} would indicate that S depends on T with a uniform distance vector $-\mathbf{d}$.

A version of Algorithm 5.1 for this restricted dependence problem is stated below:

Algorithm 5.2 Let $\mathbf{L} = (L_1, L_2, \dots, L_m)$ denote a regular loop nest with index vector \mathbf{I} , lower limit vector \mathbf{p}_0 , lower limit matrix \mathbf{P} , upper limit vector \mathbf{q}_0 , and upper limit matrix \mathbf{P} . Let S and T denote two statements in \mathbf{L} such that $S \leq T$. Let $X(\mathbf{IA} + \mathbf{a}_0)$ denote a variable of S and $X(\mathbf{IA} + \mathbf{b}_0)$ a variable of T , where X is an n -dimensional array, \mathbf{A} is an $m \times n$ integer matrix, and \mathbf{a}_0 and \mathbf{b}_0 are integer n -vectors.

Given the lexical ordering of S and T , the vectors \mathbf{p}_0 , \mathbf{q}_0 , \mathbf{a}_0 , \mathbf{b}_0 , and the matrices \mathbf{P} and \mathbf{A} , this algorithm gives the major steps of a process that decides if the two variables cause a dependence between S and T . In case they do cause a dependence, we also show how to compute the dependence of T on S , the dependence of S on T , and the distance vectors, direction vectors, and levels of each dependence.²

1. By Algorithm 2.1, find an $m \times m$ unimodular matrix \mathbf{U} and an $m \times n$ echelon matrix \mathbf{S} , such that $\mathbf{UA} = \mathbf{S}$.
2. Solve the system of diophantine equations:

$$\mathbf{tS} = \mathbf{b}_0 - \mathbf{a}_0 \quad (5.25)$$

and find the general solution \mathbf{t} if a solution exists.

[This system is easy to solve since \mathbf{S} is an echelon matrix.]

3. (Generalized GCD Test) If there is no (integer) solution to (5.25), then there is no dependence between statements S and T , and the algorithm is terminated.
4. Set

$$\begin{aligned} \rho &\leftarrow \text{rank}(\mathbf{S}) \\ M &\leftarrow m - \rho. \end{aligned}$$

²Note that here the sizes of the matrices \mathbf{U} and \mathbf{S} , and of the vector \mathbf{t} are different from their sizes in Algorithm 5.1. We still define ρ to be the rank of \mathbf{S} , but $M = m - \rho$ since \mathbf{t} now has m components.

[Note that ρ is the number of nonzero rows of \mathbf{S} . The first ρ components of \mathbf{t} are known integers, and the last M components are undetermined.]

5. [By Theorem 3.6, the general solution to the equation $\mathbf{dA} = \mathbf{a}_0 - \mathbf{b}_0$ is $\mathbf{d} = \mathbf{tU}$. Substitute for \mathbf{d} in (5.24) to get the set of inequalities in the undetermined components of \mathbf{t} .]

If the system of $2m$ inequalities

$$\mathbf{p}_0 - \mathbf{q}_0 \leq \mathbf{t}(\mathbf{UP}) \leq \mathbf{q}_0 - \mathbf{p}_0 \quad (5.26)$$

in M variables (the undetermined components of \mathbf{t}) has no (integer) solution, then there is no dependence between the statements S and T , and the algorithm is terminated.

6. Select case based on the lexical ordering of S and T :

Case ($S < T$): Assign to the set of distance vectors for the dependence of T on S the set

$$\{\mathbf{d} : \mathbf{d} = \mathbf{tU}\}$$

where \mathbf{t} is an arbitrary solution to (5.25) that satisfies (5.26) and $\mathbf{tU} \succeq \mathbf{0}$.

Assign to the set of distance vectors for the dependence of S on T the set

$$\{\mathbf{d} : \mathbf{d} = -\mathbf{tU}\}$$

where \mathbf{t} is an arbitrary solution to (5.25) that satisfies (5.26) and $\mathbf{tU} \prec \mathbf{0}$.

Case ($S = T$): Assign to the set of distance vectors for the dependence of S on itself the set

$$\{\mathbf{d} : \mathbf{d} = \mathbf{tU}\}$$

where \mathbf{t} is an arbitrary solution to (5.25) that satisfies (5.26) and $\mathbf{tU} \succ \mathbf{0}$.

In either case, if the set of distances for the dependence of T on S is empty, then T does not depend on S . If the set of distances for the dependence of S on T is empty, then S does not depend on T .

7. If T depends on S , then assign to the dependence of T on S the set of all instance pairs of the form $(S(\mathbf{i}), T(\mathbf{i} + \mathbf{d}))$ where \mathbf{d} is any distance (of the same dependence), and \mathbf{i} and $\mathbf{i} + \mathbf{d}$ are index points (i.e., $\mathbf{p}_0 \leq \mathbf{i} \leq \mathbf{q}_0$ and $\mathbf{p}_0 - \mathbf{d} \leq \mathbf{i} \leq \mathbf{q}_0 - \mathbf{d}$).

If S depends on T , then assign to the dependence of S on T the set of all instance pairs of the form $(T(\mathbf{j}), S(\mathbf{j} + \mathbf{d}))$ where \mathbf{d} is any distance (of the same dependence), and \mathbf{j} and $\mathbf{j} + \mathbf{d}$ are index points (i.e., $\mathbf{p}_0 \leq \mathbf{j} \leq \mathbf{q}_0$ and $\mathbf{p}_0 - \mathbf{d} \leq \mathbf{j} \leq \mathbf{q}_0 - \mathbf{d}$).

8. For each distance vector \mathbf{d} of a given dependence, compute the corresponding direction vector $\mathbf{sig}(\mathbf{d})$ and the corresponding level $\text{lev}(\mathbf{d})$ (of the same dependence).
9. Terminate the algorithm. □

We will end this section with some examples that illustrate uniform distance vectors in a rectangular or regular double loop. In each example, the common coefficient matrix \mathbf{A} has a rank of m or $m - 1$, so that the value of M is 0 or 1, and the system of inequalities (5.26) is easily solvable. When $M \geq 2$, it becomes more difficult to solve this system; we will not discuss that case.

Example 5.8 In the rectangular double loop

```

 $L_1 :$       do  $I_1 = 10, 200$ 
 $L_2 :$       do  $I_2 = 7, 167$ 
 $S :$            $X(2I_1 + 3, 5I_2 - 1) = \dots$ 
 $T :$            $\dots = \dots X(2I_1 - 1, 5I_2 - 6) \dots$ 
            enddo
        enddo
    
```

we have

$$\mathbf{P} = \mathcal{I}_2, \mathbf{p}_0 = (10, 7), \mathbf{Q} = \mathcal{I}_2, \mathbf{q}_0 = (200, 167),$$

and

$$\mathbf{A} = \begin{pmatrix} 2 & 0 \\ 0 & 5 \end{pmatrix}, \mathbf{a}_0 = (3, -1), \mathbf{B} = \begin{pmatrix} 2 & 0 \\ 0 & 5 \end{pmatrix}, \mathbf{b}_0 = (-1, -6).$$

Thus, $\mathbf{P} = \mathbf{Q}$ and $\mathbf{A} = \mathbf{B}$, so that Algorithm 5.2 can be applied.

The dependence equation $\mathbf{d}\mathbf{A} = \mathbf{a}_0 - \mathbf{b}_0$ becomes

$$(d_1, d_2) \begin{pmatrix} 2 & 0 \\ 0 & 5 \end{pmatrix} = (4, 5)$$

which has a unique solution: $(d_1, d_2) = (2, 1)$. The dependence constraints

$$\mathbf{p}_0 - \mathbf{q}_0 \leq \mathbf{d}\mathbf{P} \leq \mathbf{q}_0 - \mathbf{p}_0$$

become

$$(-190, -160) \leq (2, 1) \leq (190, 160)$$

which are trivially satisfied. Thus, there is one and only one distance vector in this case: $(2, 1)$, and it is uniform. For any index point (i_1, i_2) such that $(i_1 + 2, i_2 + 1)$ is also an index point, the instance $T(i_1 + 2, i_2 + 1)$ of statement T depends on the instance $S(i_1, i_2)$ of statement S . The restrictions on i_1 and i_2 are $10 \leq i_1 \leq 198$ and $1 \leq i_2 \leq 166$. The only direction vector for this dependence is $(1, 1)$. Since there is no negative \mathbf{d} , statement S does not depend on statement T .

Example 5.9 In the rectangular double loop

```

 $L_1 : \quad \text{do } I_1 = 1, 1000$ 
 $L_2 : \quad \quad \quad \text{do } I_2 = 71, 300$ 
 $S : \quad \quad \quad X(2I_1 + 3I_2 - 2, 3I_1 + 4I_2 + 1) = \dots$ 
 $T : \quad \quad \quad \dots = \dots X(2I_1 + 3I_2 + 1, 3I_1 + 4I_2 + 3) \dots$ 
 $\quad \quad \quad \text{enddo}$ 
 $\quad \quad \quad \text{enddo}$ 

```

we have

$$\mathbf{P} = \mathcal{I}_2, \mathbf{p}_0 = (1, 71), \mathbf{Q} = \mathcal{I}_2, \mathbf{q}_0 = (1000, 300),$$

and

$$\mathbf{A} = \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix}, \mathbf{a}_0 = (-2, 1), \mathbf{B} = \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix}, \mathbf{b}_0 = (1, 3).$$

Thus, $\mathbf{P} = \mathbf{Q}$ and $\mathbf{A} = \mathbf{B}$, so that Algorithm 5.2 can be applied.

The dependence equation $\mathbf{d}\mathbf{A} = \mathbf{a}_0 - \mathbf{b}_0$ becomes

$$(d_1, d_2) \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix} = (-3, -2).$$

Solving it in the usual way (Theorem 3.6 and Algorithm 2.1), we find that there is a unique solution: $(d_1, d_2) = (6, -5)$. The dependence constraints

$$\mathbf{p}_0 - \mathbf{q}_0 \leq \mathbf{d}\mathbf{P} \leq \mathbf{q}_0 - \mathbf{p}_0$$

become

$$(-999, -229) \leq (6, -5) \leq (999, 229)$$

which are trivially satisfied. Thus, there is one and only one distance vector in this case: $(6, -5)$, and it is uniform. For any index point (i_1, i_2) such that $(i_1 + 6, i_2 - 5)$ is also an index point, the instance $T(i_1 + 6, i_2 - 5)$ of statement T depends on the instance $S(i_1, i_2)$ of statement S . The restrictions on i_1 and i_2 are $1 \leq i_1 \leq 994$ and $76 \leq i_2 \leq 300$. The only direction vector for this dependence is $(1, -1)$. Since there is no negative \mathbf{d} , statement S does not depend on statement T .

Example 5.10 Consider the regular double loop:

```

 $L_1 :$       do  $I_1 = 0, 100$ 
 $L_2 :$           do  $I_2 = I_1, I_1 + 50$ 
 $S :$              $X(2I_1 + 3I_2 + 12) = \dots$ 
 $T :$              $\dots = \dots X(2I_1 + 3I_2 - 5) \dots$ 
                  enddo
              enddo

```

In Algorithm 5.2, if $M = 0$ then either there is a unique solution \mathbf{d} to the dependence equation, or no solutions at all. We had $M = 0$ in the previous two examples. In this example, the common coefficient

matrix is chosen to have a rank of $m - 1$, so that $M = 1$. We will see that there are several uniform distance vectors in this case.

Here we have

$$\mathbf{P} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}, \mathbf{p}_0 = (0, 0), \mathbf{Q} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}, \mathbf{q}_0 = (100, 50),$$

and

$$\mathbf{A} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \mathbf{a}_0 = (12), \mathbf{B} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \mathbf{b}_0 = (-5).$$

Thus, $\mathbf{P} = \mathbf{Q}$ and $\mathbf{A} = \mathbf{B}$, so that Algorithm 5.2 can be applied.

The dependence equation $\mathbf{d}\mathbf{A} = \mathbf{a}_0 - \mathbf{b}_0$ becomes

$$(d_1, d_2) \begin{pmatrix} 2 \\ 3 \end{pmatrix} = 17.$$

Solving it in the usual way (Theorem 3.6 and Algorithm 2.1), we find the general solution:

$$(d_1, d_2) = (-17 + 3t, 17 - 2t)$$

where t is an arbitrary integer. The dependence constraints

$$\mathbf{p}_0 - \mathbf{q}_0 \leq \mathbf{d}\mathbf{P} \leq \mathbf{q}_0 - \mathbf{p}_0$$

become

$$(-100, -50) \leq (-17 + 3t, 34 - 5t) \leq (100, 50)$$

which give the following range for t : $-3 \leq t \leq 16$. Thus, the set of solutions \mathbf{d} to the dependence equation that also satisfy the constraints is

$$\{(d_1, d_2) : d_1 = -17 + 3t, d_2 = 17 - 2t, -3 \leq t \leq 16\}. \quad (5.27)$$

A (lexicographically) nonnegative \mathbf{d} , if any, would be a distance vector for the dependence of T on S , while a negative \mathbf{d} , if any, would be a distance vector for the dependence of S on T . First, we look for a value of \mathbf{d} such that $d_1 > 0$. We have $d_1 > 0$ iff

$-17 + 3t > 0$, or $t \geq 6$. Thus, T depends on S at level 1, and for this dependence, the distance vectors are

$$\{(-17 + 3t, 17 - 2t) : 6 \leq t \leq 16\} = \{(1, 5), (4, 3), \dots, (31, -15)\}.$$

In fact, these are all the distance vectors for the dependence of T on S . Since there is no (integral) value of t for which $d_1 = 0$, statement T does not depend on statement S at level 2 or 3. It is easy to classify the distance vectors at level 1 according to their direction vector. The distance vectors for the dependence with direction vector $(1, 1)$ are given by $6 \leq t \leq 8$, and those for the dependence with direction vector $(1, -1)$ are given by $9 \leq t \leq 16$, while T does not depend on S with the direction vector $(1, 0)$.

Similarly, it can be shown that S depends on T at level 1 with the distance vectors

$$\begin{aligned} & \{(-d_1, -d_2) : d_1 = -17 + 3t, d_2 = 17 - 2t, -3 \leq t \leq 5\} \\ &= \{(26, -23), (29, -25), \dots, (2, -7)\}. \end{aligned}$$

They all correspond to the direction vector $(1, -1)$.

EXERCISES 5.5

1. Assume that \mathbf{p}_0 and \mathbf{q}_0 are integer m -vectors with $\mathbf{p}_0 \leq \mathbf{q}_0$. Show that given any integer vector \mathbf{x} satisfying

$$\mathbf{p}_0 - \mathbf{q}_0 \leq \mathbf{x} \leq \mathbf{q}_0 - \mathbf{p}_0,$$

there is always an integer vector \mathbf{y} such that

$$\left. \begin{array}{lcl} \mathbf{p}_0 & \leq & \mathbf{y} & \leq & \mathbf{q}_0 \\ \mathbf{p}_0 - \mathbf{x} & \leq & \mathbf{y} & \leq & \mathbf{q}_0 - \mathbf{x}. \end{array} \right\} \quad (5.28)$$

Using this, prove that if \mathbf{d} is a solution to (5.23) satisfying (5.24), then there is a solution $(\mathbf{i}; \mathbf{j})$ to (5.22) satisfying (5.21) such that $\mathbf{j} = \mathbf{i} + \mathbf{d}$.

2. Take an ideal loop nest \mathbf{L} whose index space is all of \mathbf{Z}^m . Assume that a variable $X(\mathbf{IA} + \mathbf{a}_0)$ of a statement S and a variable $X(\mathbf{IB} + \mathbf{b}_0)$ of a statement T cause a dependence of T on S . Prove that
- The dependence has a uniform distance iff $\mathbf{A} = \mathbf{B}$.
 - If the dependence has one uniform distance, then each distance is uniform.

3. In Example 5.10, for each distance vector (d_1, d_2) of the dependence of T on S , find the set of index points (i_1, i_2) such that $T(i_1 + d_1, i_2 + d_2)$ depends on $S(i_1, i_2)$. Do the same for the dependence of S on T .
4. Repeat Example 5.10 and Exercise 3 after changing the loop limits as follows: $-100 \leq I_1 \leq 200$, $0 \leq I_2 \leq 200$.
5. Find the distance vectors of each dependence between S and T in the loop:

```

do  $I = p, q$ 
 $S : X(aI + a_0) = \dots$ 
 $T : \dots = \dots X(aI + b_0) \dots$ 
enddo

```

where

- (a) $p = 0, q = 100, a = 3, a_0 = 1, b_0 = 3$
- (b) $p = 0, q = 100, a = 3, a_0 = 1, b_0 = 4$
- (c) $p = -10, q = 120, a = 2, a_0 = -4, b_0 = 2$.
6. Find conditions on p, q, a, a_0, b_0 in the loop of the previous exercise, so that there may be a dependence of T on S . Also, find conditions so that there may be a dependence of S on T .
7. Find the distance vectors of each dependence between S and T in the loop nest

```

do  $I_1 = 1, 1000$ 
do  $I_2 = 32, 564$ 
 $S : u = \dots$ 
 $T : \dots = \dots v \dots$ 
enddo
enddo

```

where

- (a) $u = X(I_1 + 1, 2I_2 + 2), v = X(I_1 + 3, 2I_2 + 9)$
- (b) $u = X(2I_1 + 3I_2 + 1, 2I_1 + I_2),$
 $v = X(2I_1 + 3I_2 + 5, 2I_1 + I_2 + 8)$
- (c) $u = X(I_1 + I_2 + 6, 2I_1 + 2I_2 + 12),$
 $v = X(I_1 + I_2 + 1, 2I_1 + 2I_2 + 2).$
8. Repeat the previous exercise after changing the loop limits as follows:
 $0 \leq I_1 \leq 200, 2I_1 + 20 \leq I_2 \leq 2I_1 + 220.$

9. Find conditions on $p_1, q_1, p_2, q_2, a_1, a_2, a_0, b_0$ such that there may be a dependence of T on S , or of S on T in the program³

```

do  $I_1 = p_1, q_1$ 
  do  $I_2 = p_2, q_2$ 
     $S :$        $X(a_1I_1 + a_2I_2 + a_0) = \dots$ 
     $T :$        $\dots = \dots X(a_1I_1 + a_2I_2 + b_0) \dots$ 
    enddo
  enddo

```

10. Find the distance vectors of each dependence between S and T in the loop nest

```

do  $I_1 = 0, 100$ 
  do  $I_2 = I_1, I_1 + 200$ 
    do  $I_3 = 2I_2, 2I_2 + 150$ 
       $S :$        $u = \dots$ 
       $T :$        $\dots = \dots v \dots$ 
    enddo
  enddo
enddo

```

where

- (a) $u = X(3I_3 - 1, I_2, I_1)$, $v = X(3I_3 + 5, I_2 - 2, I_1 + 1)$
- (b) $u = X(2I_1 - 3I_2, I_1 + I_2 + 3I_3 - 4, 2I_2 + 3I_3 - 1)$,
 $v = X(2I_1 - 3I_2 + 3, I_1 + I_2 + 3I_3 - 1, 2I_2 + 3I_3 + 5)$
- (c) $u = X(2I_1 - 3I_2, I_1 + I_2 + 3I_3 - 4, 3I_1 + I_2 - 1)$,
 $v = X(2I_1 - 3I_2 + 3, I_1 + I_2 + 3I_3 - 1, 3I_1 + I_2 + 5)$.

5.6 Rectangular Loop Nest

We assume in this section that the loop nest \mathbf{L} is rectangular, that is, the loop limits p_r and q_r are integer constants. Then we have

$$\mathbf{p}_0 = (p_1, p_2, \dots, p_m), \mathbf{P} = \mathcal{I}_m, \mathbf{q}_0 = (q_1, q_2, \dots, q_m), \mathbf{Q} = \mathcal{I}_m.$$

The index space of \mathbf{L} consists of all integer m -vectors \mathbf{I} that satisfy

$$\mathbf{p}_0 \leq \mathbf{I} \leq \mathbf{q}_0.$$

³Jay Hoeflinger of the University of Illinois has studied a large class of dependence problems of this type. He and I have had several discussions on some of his problems.

The dependence constraints are simpler than in the general case:

$$\left. \begin{array}{l} p_0 \leq i \leq q_0 \\ p_0 \leq j \leq q_0 \end{array} \right\} \quad (5.29)$$

Consider the restricted dependence problem where the array X is m -dimensional, and the coefficient matrices of the variables $X(\mathbf{IA} + \mathbf{a}_0)$ and $X(\mathbf{IB} + \mathbf{b}_0)$ are $m \times m$ diagonal matrices:

$$\mathbf{A} = \begin{pmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{mm} \end{pmatrix}, \mathbf{B} = \begin{pmatrix} b_{11} & & & \\ & b_{22} & & \\ & & \ddots & \\ & & & b_{mm} \end{pmatrix}.$$

Write

$$\mathbf{b}_0 - \mathbf{a}_0 = \mathbf{c} = (c_1, c_2, \dots, c_m).$$

Then the dependence equation

$$\mathbf{iA} - \mathbf{jB} = \mathbf{b}_0 - \mathbf{a}_0 \quad (5.30)$$

is equivalent to a system of independent two-variable equations:

$$a_{rr}i_r - b_{rr}j_r = c_r \quad (1 \leq r \leq m). \quad (5.31)$$

In terms of the components, the dependence constraints (5.29) can be written as

$$\left. \begin{array}{l} p_r \leq i_r \leq q_r \\ p_r \leq j_r \leq q_r \end{array} \right\} \quad (5.32)$$

for $1 \leq r \leq m$. It is clear that solving (5.30) subject to (5.29) is equivalent to separately solving m independent two-variable equations each with its own constraints: For $1 \leq r \leq m$, solve the r^{th} equation in (5.31) subject to the constraints (5.32).

Algorithm 5.3 which solves this dependence problem is based on repeated applications of Algorithm 3.1. Before presenting Algorithm 5.3, we will illustrate the process by one example.

Example 5.11 Consider the dependence problem in the loop nest:

```

 $L_1 :$       do  $I_1 = 0, 35$ 
 $L_2 :$           do  $I_2 = 0, 35$ 
 $S :$              $X(6I_1 - 1, 2I_2 - 1) = \dots$ 
 $T :$              $\dots = \dots X(4I_1 + 9, 2I_2 + 9) \dots$ 
        enddo
    enddo
  
```

Here we have

$$\mathbf{A} = \begin{pmatrix} 6 & 0 \\ 0 & 2 \end{pmatrix}, \mathbf{a}_0 = (-1, -1), \mathbf{B} = \begin{pmatrix} 4 & 0 \\ 0 & 2 \end{pmatrix}, \mathbf{b}_0 = (9, 9),$$

so that the coefficient matrices are diagonal. The dependence equation (5.30)

$$(i_1, i_2) \begin{pmatrix} 6 & 0 \\ 0 & 2 \end{pmatrix} - (j_1, j_2) \begin{pmatrix} 4 & 0 \\ 0 & 2 \end{pmatrix} = (10, 10) \quad (5.33)$$

is equivalent to two independent scalar equations:

$$6i_1 - 4j_1 = 10 \quad (5.34)$$

$$2i_2 - 2j_2 = 10. \quad (5.35)$$

The dependence constraints are:

$$0 \leq i_1 \leq 35 \text{ and } 0 \leq j_1 \leq 35 \quad (5.36)$$

$$0 \leq i_2 \leq 35 \text{ and } 0 \leq j_2 \leq 35. \quad (5.37)$$

Let Ψ_1 denote the set of all solutions (i_1, j_1) to (5.34) that satisfy (5.36), and $\Psi_{1,1}$, $\Psi_{1,-1}$, $\Psi_{1,0}$ the following subsets of Ψ_1 :

$$\begin{aligned} \Psi_{1,1} &= \{(i_1, j_1) \in \Psi_1 : i_1 < j_1\} \\ \Psi_{1,-1} &= \{(i_1, j_1) \in \Psi_1 : i_1 > j_1\} \\ \Psi_{1,0} &= \{(i_1, j_1) \in \Psi_1 : i_1 = j_1\}. \end{aligned}$$

Also, let Ψ_2 denote the set of all solutions (i_2, j_2) to (5.35) that satisfy (5.37), and $\Psi_{2,1}$, $\Psi_{2,-1}$, $\Psi_{2,0}$ the subsets of Ψ_2 defined in the

same way. From Example 3.5, we have:

$$\begin{aligned}\Psi_1 &= \{(5 + 2t_1, 5 + 3t_1) : -1 \leq t_1 \leq 10\} \\ \Psi_{1,1} &= \{(5 + 2t_1, 5 + 3t_1) : 1 \leq t_1 \leq 10\} \\ \Psi_{1,-1} &= \{(3, 2)\} \\ \Psi_{1,0} &= \{(5, 5)\},\end{aligned}$$

and from Example 3.6, we have:

$$\begin{aligned}\Psi_2 &= \{(t_2, -5 + t_2) : 5 \leq t_2 \leq 35\} \\ \Psi_{2,1} &= \emptyset \\ \Psi_{2,-1} &= \{(t_2, -5 + t_2) : 5 \leq t_2 \leq 35\} \\ \Psi_{2,0} &= \emptyset.\end{aligned}$$

Suppose we want to determine if statement T depends on statement S with the direction vector $(1, -1)$. This will happen if there is a solution $((i_1, i_2), (j_1, j_2))$ to the dependence equation (5.33) satisfying (5.36) and (5.37), such that $i_1 < j_1$ and $i_2 > j_2$, that is, $(i_1, j_1) \in \Psi_{1,1}$ and $(i_2, j_2) \in \Psi_{2,-1}$. Since $\Psi_{1,1}$ and $\Psi_{2,-1}$ are both nonempty, such solutions exist, and therefore T does depend on S with this direction vector. The dependence of T on S with direction vector $(1, -1)$ is the set of all instance pairs $(S(i_1, i_2), T(j_1, j_2))$ such that $T(j_1, j_2)$ depends on $S(i_1, i_2)$ and $i_1 < j_1$, $i_2 > j_2$. This set is given by

$$i_1 = 5 + 2t_1, j_1 = 5 + 3t_1, i_2 = t_2, j_2 = -5 + t_2$$

where $1 \leq t_1 \leq 10$ and $5 \leq t_2 \leq 35$. The total number of these instance pairs is 310. The set of distance vectors for this dependence is the set of possible values of $(j_1 - i_1, j_2 - i_2)$, that is,

$$\{(t_1, -5) : 1 \leq t_1 \leq 10\} = \{(1, -5), (2, -5), \dots, (10, -5)\}.$$

Whether or not T depends on S with a direction vector other than $(1, -1)$ can be decided in a similar way. Next, we consider the dependence of S on T . Note that now the roles of the index points (i_1, i_2) and (j_1, j_2) are reversed: the instance $T(j_1, j_2)$ must come before the instance $S(i_1, i_2)$ in the execution of \mathbf{L} .

Suppose we want to decide if S depends on T with the direction vector $(1, 1)$. This will happen if there is a solution $((i_1, i_2), (j_1, j_2))$ to the dependence equation (5.33) satisfying (5.36) and (5.37), such that $j_1 < i_1$ and $j_2 < i_2$, or $i_1 > j_1$ and $i_2 > j_2$, that is, $(i_1, j_1) \in \Psi_{1,-1}$ and $(i_2, j_2) \in \Psi_{2,-1}$. Since $\Psi_{1,-1}$ and $\Psi_{2,-1}$ are both nonempty, such solutions exist, and therefore S does depend on T with this direction vector. The dependence of S on T with direction vector $(1, 1)$ is the set of all instance pairs $(T(j_1, j_2), S(i_1, i_2))$ such that $S(i_1, i_2)$ depends on $T(j_1, j_2)$ and $i_1 > j_1$, $i_2 > j_2$. This set is given by

$$i_1 = 3, j_1 = 2, i_2 = t_2, j_2 = -5 + t_2$$

where $5 \leq t_2 \leq 35$. The total number of these instance pairs is 31. The set of distance vectors for this dependence is the set of possible values of $(i_1 - j_1, i_2 - j_2)$; it is the singleton set $\{(1, 5)\}$.

Whether or not S depends on T with a direction vector other than $(1, 1)$ can be decided in a similar way.

Algorithm 5.3 Let $\mathbf{L} = (L_1, L_2, \dots, L_m)$ denote a rectangular loop nest with index vector \mathbf{I} , lower limit vector \mathbf{p}_0 , and upper limit vector \mathbf{q}_0 . Let S and T denote statements in \mathbf{L} such that $S \leq T$. Let $X(\mathbf{IA} + \mathbf{a}_0)$ denote a variable of S and $X(\mathbf{IB} + \mathbf{b}_0)$ a variable of T , where X is an m -dimensional array, \mathbf{A} and \mathbf{B} are $m \times m$ (integer) diagonal matrices, and \mathbf{a}_0 and \mathbf{b}_0 are (integer) m -vectors.

Given the lexical ordering of S and T , the vectors

$$\begin{aligned}\mathbf{p}_0 &= (p_1, p_2, \dots, p_m) \\ \mathbf{q}_0 &= (q_1, q_2, \dots, q_m) \\ \mathbf{a}_0 &= (a_{10}, a_{20}, \dots, a_{m0}) \\ \mathbf{b}_0 &= (b_{10}, b_{20}, \dots, b_{m0}).\end{aligned}$$

and the main diagonals $(a_{11}, a_{22}, \dots, a_{mm})$ and $(b_{11}, b_{22}, \dots, b_{mm})$ of the matrices \mathbf{A} and \mathbf{B} , respectively, this algorithm

- Decides if the two variables $X(\mathbf{IA} + \mathbf{a}_0)$ of S and $X(\mathbf{IB} + \mathbf{b}_0)$ of T cause a dependence of T on S and/or a dependence of S on T ;

- For each nonnegative direction vector σ of size m ,
 - Decides if T depends on S with σ ;
 - Finds the dependence of T on S with σ , that is, the set of all instance pairs $(S(\mathbf{i}), T(\mathbf{j}))$ such that $T(\mathbf{j})$ depends on $S(\mathbf{i})$ and $\text{sig}(\mathbf{j} - \mathbf{i}) = \sigma$;
 - Finds the set of all distance vectors for this dependence;
 - Decides if S depends on T with σ ;
 - Finds the dependence of S on T with σ , that is, the set of all instance pairs $(T(\mathbf{j}), S(\mathbf{i}))$ such that $S(\mathbf{i})$ depends on $T(\mathbf{j})$ and $\text{sig}(\mathbf{i} - \mathbf{j}) = \sigma$;
 - Finds the set of all distance vectors for this dependence.

1. Set

$$(c_1, c_2, \dots, c_m) \leftarrow (b_{10} - a_{10}, b_{20} - a_{20}, \dots, b_{m0} - a_{m0}).$$

2. For $r = 1, 2, \dots, m$, use Algorithm 3.1 to find

- (a) The set Ψ_r of all (integer) solutions (i_r, j_r) to the diophantine equation

$$a_{rr}i_r - b_{rr}j_r = c_r \quad (5.38)$$

satisfying $p_r \leq i_r \leq q_r, p_r \leq j_r \leq q_r$.

- (b) The following subsets of Ψ_r :

$$\begin{aligned}\Psi_{r,1} &= \{(i_r, j_r) \in \Psi_r : i_r < j_r\} \\ \Psi_{r,-1} &= \{(i_r, j_r) \in \Psi_r : i_r > j_r\} \\ \Psi_{r,0} &= \{(i_r, j_r) \in \Psi_r : i_r = j_r\}.\end{aligned}$$

3. If $\Psi_r = \emptyset$ for any r , then there is no dependence between S and T , and the algorithm is terminated.
4. Repeat this step for each (lexicographically) positive direction vector $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_m)$ of size m :

- 4.1. If $\Psi_{r,\sigma_r} = \emptyset$ for any r in $1 \leq r \leq m$, then skip this step. Otherwise, statement T depends on statement S with direction vector σ . The dependence of T on S with σ consists of all pairs of statement instances $(S(\mathbf{i}), T(\mathbf{j}))$ such that $(i_r, j_r) \in \Psi_{r,\sigma_r}$ for each r in $1 \leq r \leq m$. The set of distance vectors for this dependence consists of all vectors $\mathbf{d} = (d_1, d_2, \dots, d_m)$ such that $d_r = j_r - i_r$ for some $(i_r, j_r) \in \Psi_{r,\sigma_r}$, $1 \leq r \leq m$.
- 4.2. If $\Psi_{r,-\sigma_r} = \emptyset$ for any r in $1 \leq r \leq m$, then skip this step. Otherwise, statement S depends on statement T with direction vector σ . The dependence of S on T with σ consists of all pairs of statement instances $(T(\mathbf{j}), S(\mathbf{i}))$ such that $(i_r, j_r) \in \Psi_{r,-\sigma_r}$ for each r in $1 \leq r \leq m$. The set of distance vectors for this dependence consists of all vectors $\mathbf{d} = (d_1, d_2, \dots, d_m)$ such that $d_r = i_r - j_r$ for some $(i_r, j_r) \in \Psi_{r,-\sigma_r}$, $1 \leq r \leq m$.
5. If $S < T$, and $\Psi_{r,0} \neq \emptyset$ for each r in $1 \leq r \leq m$, then T depends on S with the zero direction vector (i.e., at level $m+1$). The dependence of S on T with $\mathbf{0}$ consists of all pairs of statement instances $(S(\mathbf{i}), T(\mathbf{i}))$ such that $(i_r, i_r) \in \Psi_{r,0}$ for each r in $1 \leq r \leq m$.
6. If no direction vector σ (with $\sigma \succeq \mathbf{0}$) was found such that T depends on S with σ , then T does not depend on S ; and similarly for the dependence of S on T .
7. Terminate the algorithm. □

The set of all index value pairs (\mathbf{i}, \mathbf{j}) that define the dependence of T on S with a given direction vector σ can be expressed in terms of the individual sets Ψ_{r,σ_r} by a simple formula. Let U_1, U_2, \dots, U_m denote sets of ordered pairs of integers. Define $U_1 \diamond U_2 \diamond \dots \diamond U_m$ to be the set

$$\{((i_1, i_2, \dots, i_m), (j_1, j_2, \dots, j_m)) : (i_r, j_r) \in U_r \text{ for } 1 \leq r \leq m\}.$$

This set can be put into one-to-one correspondence with the Cartesian product $U_1 \times U_2 \times \cdots \times U_m$ in an obvious way. The dependence of T on S with a direction vector σ is then the set

$$\{(S(\mathbf{i}), T(\mathbf{j})) : (\mathbf{i}, \mathbf{j}) \in \Psi_{1,\sigma_1} \diamond \Psi_{2,\sigma_2} \diamond \cdots \diamond \Psi_{m,\sigma_m}\}.$$

On the other hand, the dependence of S on T with a direction vector σ is the set

$$\{(T(\mathbf{j}), S(\mathbf{i})) : (\mathbf{i}, \mathbf{j}) \in \Psi_{1,-\sigma_1} \diamond \Psi_{2,-\sigma_2} \diamond \cdots \diamond \Psi_{m,-\sigma_m}\}.$$

Note that for a given ℓ in $1 \leq \ell \leq m$, statement T depends on statement S at level ℓ iff T depends on S with at least one direction vector σ such that $\text{lev}(\sigma) = \ell$. Thus, using Algorithm 5.3, we can find out the levels at which T depends on S , and the instance pairs and the distance vectors for each given level. Similar information for the dependence of S on T can also be found. (See Exercise 2.)

It is clear that in the above algorithm, by pooling together the pieces of information obtained under different direction vectors, we would find the ‘total’ information for each dependence (of T on S , and of S on T). For example, if \mathbf{d} is a distance vector of the dependence of T on S , then it will be found as a distance vector for the dependence of T on S with the direction vector $\text{sig}(\mathbf{d})$.

Algorithm 5.3 can be extended to include coefficient matrices \mathbf{A} and \mathbf{B} that are more general than diagonal matrices. The first obvious step is to permute the columns of two diagonal matrices in the same way. For example, if in Example 5.11, we had the variables $X(2I_2 - 1, 6I_1 - 1)$ and $X(2I_2 + 9, 4I_1 + 9)$, then we could have handled the dependence problem without any extra difficulty. The scalar equations (5.34) and (5.35) would be interchanged and the constraints (5.36) and (5.37) would be interchanged, but the same equation would have the same set of constraints. In general terms, this amounts to taking two coefficient matrices \mathbf{A} and \mathbf{B} such that

$$\mathbf{A} = \mathbf{EP} \quad \text{and} \quad \mathbf{B} = \mathbf{FP}$$

where \mathbf{E} and \mathbf{F} are diagonal matrices and \mathcal{P} is a permutation matrix.

Suppose $\mathbf{A} = \mathbf{EC}$ and $\mathbf{B} = \mathbf{FC}$ where \mathbf{E} and \mathbf{F} are $m \times m$ diagonal matrices and \mathbf{C} is any $m \times m$ matrix. The dependence equation (5.30) can be written as

$$\begin{aligned}\mathbf{b}_0 - \mathbf{a}_0 &= \mathbf{iA} - \mathbf{jB} \\ &= \mathbf{iEC} - \mathbf{jFC} \\ &= (\mathbf{iE} - \mathbf{jF})\mathbf{C},\end{aligned}$$

or as

$$\mathbf{kC} = \mathbf{b}_0 - \mathbf{a}_0 \quad (5.39)$$

where

$$\mathbf{iE} - \mathbf{jF} = \mathbf{k}. \quad (5.40)$$

The dependence equation has a solution $(\mathbf{i}; \mathbf{j})$ iff equation (5.39) has a solution \mathbf{k} for which equation (5.40) has a solution $(\mathbf{i}; \mathbf{j})$. For each fixed solution \mathbf{k} to (5.39), we get an instance of the problem solved in Algorithm 5.3. We will not pursue this matter any further here, except to illustrate the extended dependence problem by an example given below. Note that there is a simple way to determine if two coefficient matrices can be broken up in the form $\mathbf{A} = \mathbf{EC}$ and $\mathbf{B} = \mathbf{FC}$; see Exercise 1.

Example 5.12 Consider the loop nest

```

do  $I_1 = 0, 100$ 
  do  $I_2 = 0, 100$ 
    do  $I_3 = 0, 100$ 
       $S : \quad X(I_1 - 4I_3, 3I_1 - 3I_2, 2I_1 - 6I_2 + 20I_3) = \dots$ 
       $T : \quad \dots = \dots X(2I_3 + 1, 2I_2 + 1, 4I_2 - 10I_3 + 2) \dots$ 
    enddo
  enddo
enddo
```

The variables of S and T have the forms $X(\mathbf{IA} + \mathbf{a}_0)$ and $X(\mathbf{IB} + \mathbf{b}_0)$,

respectively, where

$$\mathbf{A} = \begin{pmatrix} 1 & 3 & 2 \\ 0 & -3 & -6 \\ -4 & 0 & 20 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 4 \\ 2 & 0 & -10 \end{pmatrix}$$

$$\mathbf{a}_0 = (0, 0, 0), \quad \mathbf{b}_0 = (1, 1, 2).$$

The corresponding rows of the two matrices can be expressed as

$$\begin{aligned} (1, 3, 2) &= 1(1, 3, 2) \quad \text{and} \quad (0, 0, 0) = 0(1, 3, 2) \\ (0, -3, -6) &= -3(0, 1, 2) \quad \text{and} \quad (0, 2, 4) = 2(0, 1, 2) \\ (-4, 0, 20) &= 2(-2, 0, 10) \quad \text{and} \quad (2, 0, -10) = -1(-2, 0, 10). \end{aligned}$$

Hence, we can write $\mathbf{A} = \mathbf{EC}$ and $\mathbf{B} = \mathbf{FC}$, where

$$\mathbf{E} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & 2 \end{pmatrix}, \quad \mathbf{F} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{pmatrix}, \quad \mathbf{C} = \begin{pmatrix} 1 & 3 & 2 \\ 0 & 1 & 2 \\ -2 & 0 & 10 \end{pmatrix}.$$

The dependence equation is

$$i\mathbf{A} - j\mathbf{B} = \mathbf{b}_0 - \mathbf{a}_0$$

or

$$i\mathbf{E} - j\mathbf{F} = \mathbf{k} \quad (5.41)$$

where

$$\mathbf{k}\mathbf{C} = (1, 1, 2). \quad (5.42)$$

Solving equation (5.42) by using Algorithm 2.1 in the usual way, we get $\mathbf{k} = (5, -14, 2)$. Equation (5.41) is then equivalent to the system of three scalar equations:

$$\left. \begin{array}{rcl} i_1 & = & 5 \\ -3i_2 - 2j_2 & = & -14 \\ 2i_3 + j_3 & = & 2. \end{array} \right\} \quad (5.43)$$

The dependence constraints in this rectangular loop nest are

$$\left. \begin{array}{rcl} 0 \leq i_r \leq 100 \\ 0 \leq j_r \leq 100 \end{array} \right\} \quad (5.44)$$

for $1 \leq r \leq 3$.

For $r = 1, 2, 3$, let Ψ_r denote the set of all (integer) solutions (i_r, j_r) to the r^{th} equation in (5.43) satisfying the r^{th} set of constraints in (5.44). Also, let $\Psi_{r,1}$, $\Psi_{r,-1}$, $\Psi_{r,0}$ denote the following subsets of Ψ_r :

$$\begin{aligned}\Psi_{r,1} &= \{(i_r, j_r) \in \Psi_r : i_r < j_r\} \\ \Psi_{r,-1} &= \{(i_r, j_r) \in \Psi_r : i_r > j_r\} \\ \Psi_{r,0} &= \{(i_r, j_r) \in \Psi_r : i_r = j_r\}.\end{aligned}$$

By repeated applications of Algorithm 3.1, we find that

$$\begin{aligned}\Psi_{1,1} &= \{(5, j_1) : 6 \leq j_1 \leq 100\} \\ \Psi_{1,-1} &= \{(5, j_1) : 0 \leq j_1 \leq 4\} \\ \Psi_{1,0} &= \{(5, 5)\} \\ \\ \Psi_{2,1} &= \{(2, 4), (0, 7)\} \\ \Psi_{2,-1} &= \{(4, 1)\} \\ \Psi_{2,0} &= \emptyset \\ \\ \Psi_{3,1} &= \{(0, 2)\} \\ \Psi_{3,-1} &= \{(1, 0)\} \\ \Psi_{3,0} &= \emptyset.\end{aligned}$$

The dependence of T on S with the (randomly chosen) direction vector $(0, 1, -1)$ is the set of all statement instance pairs $(S(i_1, i_2, i_3), T(j_1, j_2, j_3))$ such that

$$(i_1, j_1) \in \Psi_{1,0}, (i_2, j_2) \in \Psi_{2,1}, (i_3, j_3) \in \Psi_{3,-1},$$

that is, the set

$$\{(S(5, 2, 1), T(5, 4, 0)), (S(5, 0, 1), T(5, 7, 0))\}.$$

The two distance vectors for this dependence are $(0, 2, -1)$ and $(0, 7, -1)$.

The dependence of S on T with the direction vector $(0, 1, -1)$ is the set of all statement instance pairs $(T(j_1, j_2, j_3), S(i_1, i_2, i_3))$ such that

$$(i_1, j_1) \in \Psi_{1,0}, (i_2, j_2) \in \Psi_{2,-1}, (i_3, j_3) \in \Psi_{3,1},$$

that is, the set $\{(T(5, 1, 2), S(5, 4, 0))\}$. The unique distance vector for this dependence is $(0, 3, -2)$.

Dependence information relative to other direction vectors can be found similarly (Exercise 4).

Algorithm 5.3 is an extension of the single loop dependence test in [Bane79]. Direction vectors were incorporated into the single loop test in [WoBa87]; see also [Wolf89]. Details of dependence analysis are beyond the scope of this book. [Bane88a] contains much of the published research in this area up to 1988. More recent work includes [KoKP90], [LiYZ90], [MaHL91], [Pugh91] and [WoTs92]. A number of references on dependence analysis are listed in the Bibliography; see [GoKT91] for a more comprehensive list.

EXERCISES 5.6

1. Two vectors of the same size are *proportional* if they are scalar multiples of a third vector. Thus, $(2, -4, 6)$ and $(-3, 6, -9)$ are proportional since they are multiples of the vector $(1, -2, 3)$.
 - (a) Prove that two $m \times n$ matrices \mathbf{A} and \mathbf{B} have proportional rows iff there exist $m \times m$ diagonal matrices \mathbf{E} and \mathbf{F} , and an $m \times n$ matrix \mathbf{C} such that $\mathbf{A} = \mathbf{EC}$ and $\mathbf{B} = \mathbf{FC}$.
 - (b) Explain how you will determine if \mathbf{A} and \mathbf{B} have proportional rows.
2. Modify Algorithm 5.3 so that it finds the dependence of T on S and that of S on T at each possible level.
3. Repeat Example 5.11 after interchanging the subscripts of each variable.
4. In Example 5.12, find the dependence of T on S and the dependence of S on T with direction vectors not already covered.
5. Repeat the problem of Example 5.12 after changing the loop limits as follows:

$$\begin{aligned}-100 &\leq I_1 &\leq 100 \\ -100 &\leq I_2 &\leq 200 \\ -200 &\leq I_3 &\leq 100.\end{aligned}$$

6. Apply Algorithm 5.3 to the dependence problem in the loop:

```
do  $I = p, q$ 
 $S : X(aI + a_0) = \dots$ 
 $T : \dots = \dots X(bI + b_0) \dots$ 
enddo
```

where

- (a) $p = 0, q = 100, a = 3, a_0 = 1, b = 5, b_0 = 3$
- (b) $p = 0, q = 100, a = 3, a_0 = 1, b = 6, b_0 = 4$
- (c) $p = -50, q = 120, a = 2, a_0 = -4, b = -4, b_0 = 2.$

7. Apply Algorithm 5.3 to the dependence problem in the double loop:

```

do  $I_1 = 1, 1000$ 
    do  $I_2 = 0, 200$ 
         $S : \quad u = \dots$ 
         $T : \quad \dots = \dots v \dots$ 
    enddo
enddo

```

where

- (a) $u = X(I_1 + 1, 2I_2 + 2), v = X(2I_1 + 3, 2I_2 + 8)$
- (b) $u = X(2I_1 + 1, 2I_2 + 2), v = X(4I_1 + 4, 2I_2 + 8)$
- (c) $u = X(3I_2, 2I_1), v = X(4I_2 + 1, 6I_1 + 2)$
- (d) $u = X(2I_1 + 8I_2 + 1, I_1 + 6I_2 + 2),$
 $v = X(4I_1 + 12I_2 + 3, 2I_1 + 9I_2 + 4)$
- (e) $u = X(4I_1 + I_2 + 1, 6I_1 + 2I_2, 10I_1 + 3I_2),$
 $v = X(2I_1 + 3I_2 + 4, 3I_1 + 6I_2 + 4, 5I_1 + 9I_2 + 7).$

8. Describe the most general dependence problem that can be solved by both algorithms 5.2 and 5.3. Compare the two ways of solving this problem.

Part III

Loop Transformations

Chapter 6

Introduction to Loop Transformations

6.1 Introduction

Consider a sequential loop nest **L** of the form

```
L1 :    do I1 = p1, q1
L2 :        do I2 = p2, q2
          :
Lm :        do Im = pm, qm
          H(I1, I2, ..., Im)
          enddo
          :
enddo
enddo
```

where $H(I_1, I_2, \dots, I_m)$ is a sequence of assignment statements. The program consists of a set of statement instances with an implied execution order. This is a total order defined by the rules:

1. If i and j are two index values such that $i \prec j$, then the iteration $H(i)$ is to be executed before the iteration $H(j)$;
2. If S and T are two statements such that $S < T$, then in a given iteration $H(i)$, the instance $S(i)$ is to be executed before the instance $T(i)$.

To represent execution orders that cannot be defined by a sequential loop nest, we need to introduce more general program constructs.

We will extend the sequential loop nest in several steps. A **doall** loop has the form

```
doall  $I = p, q$ 
       $H(I)$ 
enddoall
```

which is like an ordinary loop except the iterations are not ordered. A perfect nest of **doall** loops is defined similarly. In such a nest, all iterations can be executed simultaneously or in any arbitrary order. A *mixed* loop nest is a nest of loops some of which are **do** loops and some are **doall** loops. Let $(L_1, \dots, L_{\ell-1}, \bar{L}_\ell, L_{\ell+1}, \dots, L_m)$ denote a mixed loop nest where the ℓ^{th} loop \bar{L}_ℓ is a **doall** loop, every other loop is a **do** loop, and the body $H(\mathbf{I})$ is a sequence of assignment statements. The execution order here is defined by the rules:

1. If \mathbf{i} and \mathbf{j} are index values such that $\mathbf{i} \prec_r \mathbf{j}$ for any r between 1 and m except ℓ , then the iteration $H(\mathbf{i})$ is to be executed before the iteration $H(\mathbf{j})$;
2. If S and T are statements in H such that $S < T$, then in a given iteration $H(\mathbf{i})$, the instance $S(\mathbf{i})$ of S is to be executed before the instance $T(\mathbf{i})$ of T .

An example of a mixed loop nest is given below:

Example 6.1 Let $H(I_1, I_2, I_3)$ denote the body of the mixed loop nest:

```
 $L_1:$     do  $I_1 = 1, 2$ 
 $\bar{L}_2:$     doall  $I_2 = 1, 2$ 
 $L_3:$         do  $I_3 = 1, 2$ 
                   $S(I_1, I_2, I_3)$ 
                   $T(I_1, I_2, I_3)$ 
                enddo
              enddoall
            enddo
```

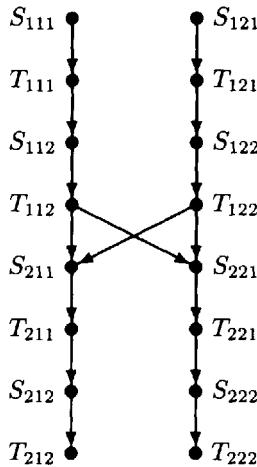


Figure 6.1: Execution order of the program in Example 6.1.

where S and T represent assignment statements. Each iteration of the form $H(1, i_2, i_3)$ is to be executed before each iteration of the form $H(2, i_2, i_3)$. Each iteration of the form $H(1, 1, i_3)$ can execute in parallel with each iteration of the form $H(1, 2, i_3)$. Each iteration of the form $H(2, 1, i_3)$ can execute in parallel with each iteration of the form $H(2, 2, i_3)$. For a fixed i_1 and a fixed i_2 , the iteration $H(i_1, i_2, 1)$ must precede the iteration $H(i_1, i_2, 2)$. In summary, an iteration $H(i_1, i_2, i_3)$ must be executed before an iteration $H(j_1, j_2, j_3)$ if

$$H(i_1, i_2, i_3) \prec_1 H(j_1, j_2, j_3) \text{ or } H(i_1, i_2, i_3) \prec_3 H(j_1, j_2, j_3).$$

The middle loop being a **doall**, $H(i_1, i_2, i_3)$ does not have to precede $H(j_1, j_2, j_3)$ when $H(i_1, i_2, i_3) \prec_2 H(j_1, j_2, j_3)$.

Within a fixed iteration $H(i_1, i_2, i_3)$, the instance $S(i_1, i_2, i_3)$ of statement S must precede the instance $T(i_1, i_2, i_3)$ of statement T . The execution order of this loop nest is represented by the acyclic graph in Figure 6.1, where we have used the notation S_{111} for $S(1, 1, 1)$ and similar notations for the other instances.

It is now clear how to define the execution order of a mixed loop nest that consists of an arbitrary mixture of **do** and **doall** loops.

Note that going from a sequential to a mixed loop nest, we have replaced an execution order that is total by an order that is only partial.

Next, consider a perfect nest of m **do** loops whose body is a *partially ordered set* (H, \leq) of assignment statements. The rules defining the execution order are formally the same as those given at the beginning of this section: $i \prec j$ implies that $H(i)$ must precede $H(j)$, and $S < T$ means $S(i)$ must precede $T(i)$. The only difference now is that there may be a pair of distinct statements S and T such that neither $S < T$, nor $T < S$. Then, for any given index value i , the instances $S(i)$ and $T(i)$ are not ordered.

The execution order of a mixed loop nest (L_1, L_2, \dots, L_m) whose body is a partially ordered set of statements (H, \leq) can now be defined by the following rules:

1. If L_r is a **do** loop, and i and j are two index values such that $i \prec_r j$, then the iteration $H(i)$ is to be executed before the iteration $H(j)$, $1 \leq r \leq m$;
2. If S and T are two statements such that $S < T$, then in a given iteration $H(i)$, the instance $S(i)$ is to be executed before the instance $T(i)$.

We may consider an acyclic digraph whose vertices are mixed loop nests of this type, and we may allow loop nests with more general bodies. A general program construct can be formally defined that would include all the special constructs introduced so far. We will not give a definition at this point, but illustrate the concept with an example:

Example 6.2 A general program of the type mentioned above is shown in Figure 6.2. The representation of the body of the loop nest, although informal, clearly shows the partial ordering of the three parts. The execution order of the statement instances of the program is shown in Figure 6.3, where we have used the subscript notation for an instance introduced in Example 6.1. (An '*' in a subscript represents a missing index variable; e.g., I_3 and I_4 are missing in statement S .)

```
doall  $I_1 = 1, 2$ 
  do  $I_2 = 1, 2$ 
```

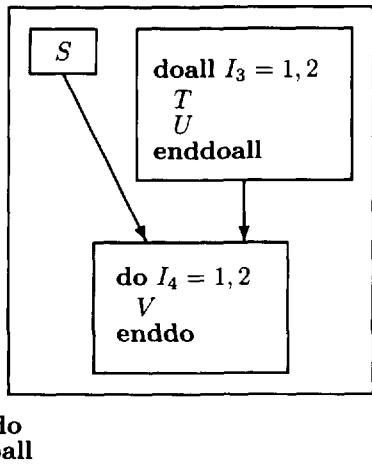


Figure 6.2: A general program.

Our object of study is a nest of **do** loops $\mathbf{L} = (L_1, L_2, \dots, L_m)$ whose body is a sequence of assignment statements. The lexical order of the statements is denoted by \leq . A *loop transformation* of \mathbf{L} is a mechanism that changes the (sequential) execution order of the statement instances without changing the set of statement instances. The result of applying a loop transformation to \mathbf{L} is a *transformed program*. A transformed program is *equivalent* to \mathbf{L} if for any two statement instances $S(i)$ and $T(j)$, $T(j)$ is forced to execute after $S(i)$ in the transformed program whenever $T(j)$ depends on $S(i)$ in \mathbf{L} .¹ A loop transformation is *valid* if the corresponding transformed program is equivalent to \mathbf{L} .

From this point, unless explicitly stated otherwise, by ‘dependence’ we will mean flow, anti-, and output dependences only. Input dependence does not play a role in the transformations we are going to study now.

¹The labels ‘ $S(i)$ ’ and ‘ $T(j)$ ’ of the instances are meaningful in \mathbf{L} , but may not be meaningful in the transformed program.

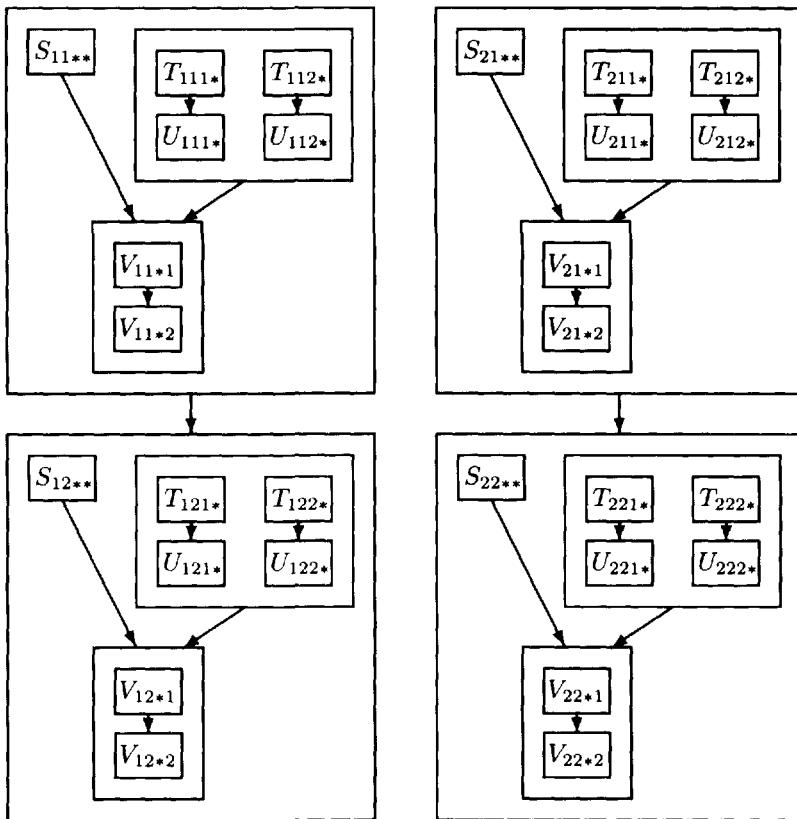


Figure 6.3: Execution order of the program in Figure 6.2.

We say that a loop L_ℓ in the nest \mathbf{L} *can execute in parallel* if the transformation that changes L_ℓ into a **doall** loop is valid. A necessary and sufficient condition for this to happen is that there be no dependence in \mathbf{L} at level ℓ (Exercise 1).

A number of loop transformations will be introduced in the following sections of this chapter. It will be done informally through examples. Our purpose here is to give a glimpse of some of the major transformations, and demonstrate how they can be implemented by using the algorithms we have developed. We will not spend any time on the dependence analysis of the program examples. Algorithm 5.2 or 5.3 would be more than adequate for the proper handling of the array subscripts which are deliberately cho-

sen to be simple. The reader should keep in mind, however, that the transformations are not restricted to these subscript forms.

We will represent the transformed programs produced by loop transformations in terms of the program constructs defined earlier in this section. For a given sequential loop nest \mathbf{L} , many equivalent transformed programs may exist. Some of those programs would exploit the characteristics of a given machine architecture better than others. For example, a vector machine expects a collection of independent instances of the same assignment statement, such that the corresponding index values form a simple pattern. On the other hand, for a shared-memory multiprocessor, one would like to have a set of iterations that are mutually independent. It may even happen that two different equivalent programs are individually suited to two different features of the same architecture.

EXERCISES 6.1

1. Prove that for $1 \leq \ell \leq m$, the loop L_ℓ in the sequential loop nest \mathbf{L} can execute in parallel iff there is no dependence in \mathbf{L} at level ℓ .
2. Represent the execution order of statement instances in the program of Example 6.1 as an acyclic digraph, after changing the program as follows:
 - (a) Change the I_1 -loop into a **doall** loop and the I_2 -loop into a **do** loop.
 - (b) Change the I_2 -loop into a **do** loop and the I_3 -loop into a **doall** loop.
3. Represent the execution order of statement instances in the program of Figure 6.2 as an acyclic digraph, after changing the program as follows:
 - (a) Change the I_1 -loop into a **do** loop and the I_2 -loop into a **doall** loop.
 - (b) Change the I_3 -loop into a **do** loop and the I_4 -loop into a **doall** loop.
 - (c) Replace S by the loop

```
doall  $I_5 = 1, 3$ 
      W
      P
enddoall
```

- (d) Replace U by the same I_5 -loop given above.

4. Explain why the following programs are or are not equivalent to the double loop

$L_1 : \text{do } I_1 = 5, 100$

$L_2 : \text{do } I_2 = 3, 100$

$$A(I_1, I_2) = A(I_1 - 1, I_2 + 1)$$

enddo

enddo

(a) **do** $I_2 = 3, 100$

do $I_1 = 5, 100$

$$A(I_1, I_2) = A(I_1 - 1, I_2 + 1)$$

enddo

enddo

(b) **do** $I_1 = 5, 100$

doall $I_2 = 3, 100$

$$A(I_1, I_2) = A(I_1 - 1, I_2 + 1)$$

enddoall

enddo

(c) **doall** $I_1 = 5, 100$

do $I_2 = 3, 100$

$$A(I_1, I_2) = A(I_1 - 1, I_2 + 1)$$

enddo

enddoall

(d) **do** $K_1 = 5, 100$

do $K_2 = 2K_1 + 3, 2K_1 + 100$

$$A(K_1, -2K_1 + K_2) = A(K_1 - 1, -2K_1 + K_2 + 1)$$

enddo

enddo

6.2 Iteration Graph Partitioning

An *iteration-level* loop transformation of \mathbf{L} is a transformation that treats individual iterations as indivisible units and changes only their order of execution. If S and T are statements such that $S < T$, then in a given iteration $H(\mathbf{i})$, the instance $S(\mathbf{i})$ is still executed before the instance $T(\mathbf{i})$. The transformations considered in sections 6.2–6.4 are all of this type.

An iteration-level loop transformation is valid if an iteration $H(\mathbf{j})$ is executed after an iteration $H(\mathbf{i})$ in the transformed pro-

gram, whenever $H(j)$ depends on $H(i)$ in \mathbf{L} . Suppose that the iteration dependence graph of \mathbf{L} has been broken up into its weakly connected components. If an iteration $H(j)$ depends on an iteration $H(i)$, then both of them must belong to the same weakly connected component. *Iteration graph partitioning* is an iteration-level loop transformation that changes the sequential execution order of \mathbf{L} into the order defined by the following rules:

1. Execute simultaneously the weakly connected components of the iteration dependence graph;
2. In each component, execute the iterations in their original (sequential) order.

This transformation is clearly valid, since if $H(j)$ depends on $H(i)$, then $H(j)$ is executed after $H(i)$ in \mathbf{L} , and the same order will be obeyed in the transformed program as the two iterations belong to the same weakly connected component.

The *dependence matrix* of \mathbf{L} is a matrix whose rows are the distance vectors of all the dependences in \mathbf{L} .² Using an analytical method based on diagonalization of the dependence matrix, we can find the decomposition of the iteration dependence graph. In the single loop case, this method reduces to a restructuring of the loop based on the gcd of the dependence distances. We illustrate the decomposition technique by two examples:

Example 6.3 Let $H(I)$ denote the body of the loop:

```
L :      do I = 0, 100
          X(I) = ...
          ... = X(I - 4) + X(I - 6)
      enddo
```

In L , there are two uniform dependence distances: 4 and 6. Thus, for each index value i , the iteration $H(i+4)$ depends on the iteration $H(i)$ if $i+4$ is an index value, and $H(i+6)$ depends on $H(i)$ if $i+6$ is

²The distance vectors can be picked in any order, so that the dependence matrix is unique up to a permutation of rows.

an index value. Note that if $H(i)$ and $H(j)$ are two iterations such that i is even and j is odd, then there can be no dependence between $H(i)$ and $H(j)$. If we split the 101 iterations into two subsequences:

$$\begin{aligned} H(0), H(2), H(4), H(6), H(8), H(10), \dots, H(100) \\ H(1), H(3), H(5), H(7), H(9), H(11), \dots, H(99), \end{aligned}$$

and simultaneously execute the subsequences, each in the given order, then all the dependence constraints will be satisfied.³ The number of subsequences is equal to the gcd of the dependence distances ($\gcd(4, 6) = 2$).

We claim that the new execution order described above is represented by the following mixed loop nest \mathbf{L}' :

```
doall  $I_1 = 0, 1$ 
  do  $I_2 = 0, \lfloor (100 - I_1)/2 \rfloor$ 
     $X(I_1 + 2I_2) = \dots$ 
     $\dots = X(I_1 + 2I_2 - 4) + X(I_1 + 2I_2 - 6)$ 
  enddo
enddoall
```

First, each integer I can be written in the form: $I = I_1 + 2I_2$ where $I_1 = 0, 1$ and I_2 is an integer. To ensure that $0 \leq I \leq 100$, we must restrict I_2 so that

$$0 \leq I_1 + 2I_2 \leq 100,$$

that is

$$\lceil -I_1/2 \rceil \leq I_2 \leq \lfloor (100 - I_1)/2 \rfloor$$

or

$$0 \leq I_2 \leq \lfloor (100 - I_1)/2 \rfloor.$$

The iterations of L are obtained from $H(I)$ by letting I run from 0 to 100. The iterations of \mathbf{L}' are obtained from $H(I_1 + 2I_2)$ by setting I_1 to 0 or 1, and then letting I_2 run from 0 to $\lfloor (100 - I_1)/2 \rfloor$. Thus, the set of iterations of \mathbf{L} is the same as the set of iterations of \mathbf{L}' . Next, the value $I_1 = 0$ gives the sequence of iterations $H(2I_2)$, $0 \leq I_2 \leq 50$, which is the first subsequence listed above. Similarly,

³These two sets of iterations constitute the two weakly connected components of the iteration dependence graph of \mathbf{L} .

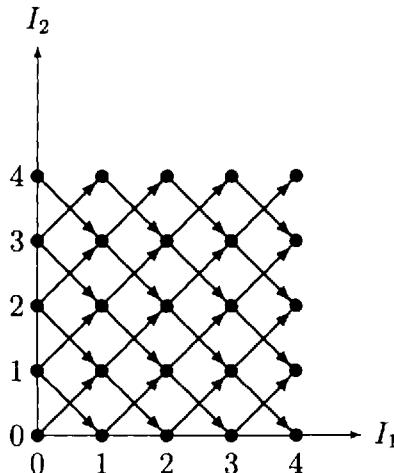


Figure 6.4: Iteration dependence graph for Example 6.4.

the value $I_1 = 1$ gives the second subsequence of iterations. Since the I_1 -loop is a **doall** loop, the two subsequences can be executed simultaneously. The iterations in each subsequence are given by the values of I_2 , and they are to be executed sequentially in their original order. As already explained, this makes the loop nest \mathbf{L}' equivalent to the given loop L .

Example 6.4 Let $H(I_1, I_2)$ denote the body of the loop nest:

```

 $L_1 :$       do  $I_1 = 0, 4$ 
 $L_2 :$           do  $I_2 = 0, 4$ 
                   $X(I_1 + 1, I_2 + 1) = Y(I_1, I_2 + 2) + 5$ 
                   $Y(I_1 + 1, I_2 + 1) = X(I_1, I_2) - 6$ 
                  enddo
enddo

```

In (L_1, L_2) , there are two uniform distance vectors: $(1, 1)$ and $(1, -1)$. The iteration dependence graph of (L_1, L_2) is shown in Figure 6.4 where we have identified an index point with the iteration it represents. Visual inspection shows that this graph has two weakly connected components; the two sets of edges are also shown separately in Figure 6.5. We will describe below how to find the

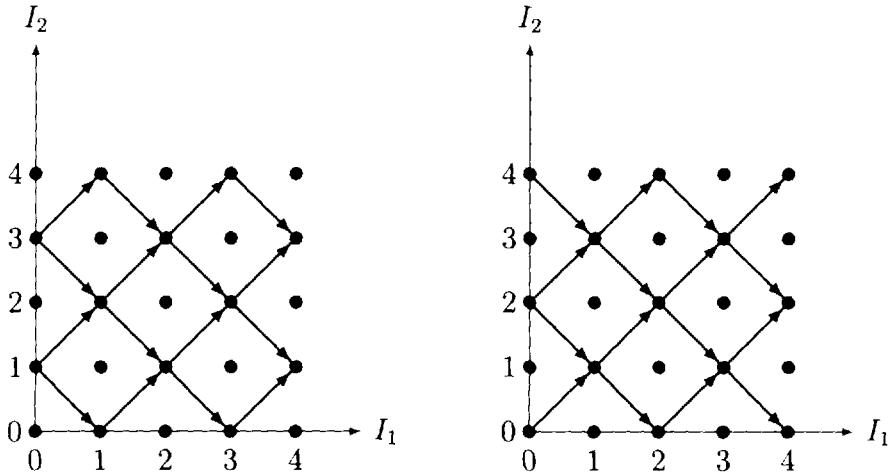


Figure 6.5: Weakly connected components of Figure 6.4.

same components in an analytical way.

We may take the dependence matrix of the loop nest to be

$$\mathcal{D} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Using Algorithm 2.3, we get the matrices

$$\mathbf{U} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}, \quad \mathbf{D} = \begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix}$$

such that \mathbf{U} and \mathbf{V} are unimodular, \mathbf{D} is diagonal, and $\mathbf{UDV} = \mathcal{D}$.

Suppose two iterations $H(i_1, i_2)$ and $H(j_1, j_2)$ are weakly connected, so that there is an undirected path in the graph joining them. The vector $(j_1, j_2) - (i_1, i_2)$ can be expressed as the vector sum of properly directed line segments lying on this path. Each such directed line segment is one of the vectors: $(1, 1)$, $-(1, 1)$, $(1, -1)$, $-(1, -1)$. Vector addition being commutative and associative, there exist integers x_1 and x_2 such that

$$(j_1, j_2) - (i_1, i_2) = x_1(1, 1) + x_2(1, -1) = (x_1, x_2)\mathcal{D}.$$

Postmultiplying both sides by \mathbf{V} , we get:

$$(j_1, j_2)\mathbf{V} - (i_1, i_2)\mathbf{V} = (x_1, x_2)\mathbf{D}\mathbf{V} = (x_1, x_2)\mathbf{U}^{-1}\mathbf{D} = (y_1, y_2)\mathbf{D}$$

where $(y_1, y_2) = (x_1, x_2)\mathbf{U}^{-1}$ is an integer vector. This gives us

$$(j_1, -j_1 + j_2) - (i_1, -i_1 + i_2) = (y_1, -2y_2)$$

or

$$\begin{aligned} j_1 - i_1 &= y_1 \\ (j_1 - j_2) - (i_1 - i_2) &= 2y_2. \end{aligned}$$

The first equation simply says that the difference $j_1 - i_1$ is an integer, which is nothing new. The second equation is significant. It implies that two iterations $H(i_1, i_2)$ and $H(j_1, j_2)$ are weakly connected iff

$$[(j_1 - j_2) - (i_1 - i_2)] \bmod 2 = 0$$

or

$$(i_1 - i_2) \bmod 2 = (j_1 - j_2) \bmod 2.$$

Thus, there are two weakly connected components. One component consists of all iterations $H(i_1, i_2)$ such that $(i_1 - i_2) \bmod 2 = 0$, and the other of all iterations such that $(i_1 - i_2) \bmod 2 = 1$.

We need to write a program consisting of the iterations of \mathbf{L} in which iterations $H(i_1, i_2)$ such that $(i_1 - i_2) \bmod 2 = 0$ can be executed in parallel with iterations $H(i_1, i_2)$ such that $(i_1 - i_2) \bmod 2 = 1$. (In each group, the iterations follow their original sequential order.) The following program construct satisfies these requirements:

```

doall  $\alpha = 0, 1$ 
  do  $I_1 = 0, 4$ 
    do  $I_2 = 0, 4$ 
      if  $(I_1 - I_2) \bmod 2 = \alpha$ 
        then
           $X(I_1 + 1, I_2 + 1) = Y(I_1, I_2 + 2) + 5$ 
           $Y(I_1 + 1, I_2 + 1) = X(I_1, I_2) - 6$ 
        endif
      enddo
    enddo
enddoall

```

It is equivalent to the given program (L_1, L_2) .⁴

The method of Example 6.3 is based on the work of David Padua [Padu79], and that of Example 6.4 is based on the work of Erik D'Hollander [D'Hol89]. (Our approach and notation are slightly different.) There is a recent paper [D'Hol92] by D'Hollander that is an extended and updated version of [D'Hol89]. Important research in this area has been done by Jih-Kwon Peir and Ron Cytron [PeCy89], and by Weijia Shang and Jose Fortes ([ShFo88], [ShFo91]).

EXERCISES 6.2

1. Apply iteration graph partitioning to the loop

```
L :      do I = 100, 300
          H(I)
        enddo
```

when the dependence distances in L are

- (a) 6 and 9
- (b) 0, 6, 9 and 12
- (c) 2 and 4
- (d) 8, 12 and 20.

Show the independent subsequences of iterations and the transformed loop nest in each case.

2. In Figure 6.4, one undirected path joining the iterations $H(0, 4)$ and $H(1, 1)$ is the sequence $(H(0, 4), H(1, 3), H(0, 2), H(1, 1))$. We can write

$$\begin{aligned}(1, 1) - (0, 4) &= [(1, 1) - (0, 2)] + [(0, 2) - (1, 3)] + [(1, 3) - (0, 4)] \\ &= (1, -1) - (1, 1) + (1, -1) \\ &= -(1, 1) + 2(1, -1).\end{aligned}$$

- (a) Find all undirected paths joining the iterations $H(0, 4)$ and $H(4, 0)$. For each path, find the corresponding linear combination of the distance vectors $(1, 1)$ and $(1, -1)$, that represents the difference $(4, 0) - (0, 4)$.
- (b) Repeat the problem for the iterations $H(0, 3)$ and $H(3, 4)$.

⁴Note that it is further possible to change the I_2 -loop into a **doall** loop (proof?).

3. Apply iteration graph partitioning to the double loop

```

 $L_1 :$       do  $I_1 = 1, 100$ 
 $L_2 :$       do  $I_2 = 1, 100$ 
             $H(I_1, I_2)$ 
            enddo
        enddo

```

where $H(I_1, I_2)$ consists of the statement

- (a) $X(I_1, I_2) = X(I_1 - 2, I_2) + X(I_1, I_2 - 3)$
- (b) $X(I_1, I_2) = X(I_1 - 3, I_2 + 3) + X(I_1, I_2 - 2) + X(I_1, I_2 - 3)$
([ShFo88] and [D'Hol92])
- (c) $X(I_1, I_2) = X(I_1 - 3, I_2 - 2).$

6.3 Unimodular Transformations

Let \mathbf{U} denote an $m \times m$ unimodular matrix. The inverse \mathbf{U}^{-1} exists and is an integer matrix. The mapping $\mathbf{I} \mapsto \mathbf{IU}$ of \mathbb{Z}^m into itself is one-to-one, since $i\mathbf{U} = j\mathbf{U}$ always implies $i = j$ (after postmultiplication by \mathbf{U}^{-1}). Also, the range of this mapping is all of \mathbb{Z}^m , since each integer m -vector \mathbf{K} is the image of the integer m -vector $\mathbf{I} = \mathbf{KU}^{-1}$.

Such a matrix \mathbf{U} can be used to define a new execution order for the iterations of \mathbf{L} in a natural way: An iteration $H(\mathbf{i})$ is to be executed before an iteration $H(\mathbf{j})$ iff $i\mathbf{U} \prec j\mathbf{U}$. This is clearly a total order, since given two distinct iterations $i\mathbf{U}$ and $j\mathbf{U}$, we must have either $i\mathbf{U} \prec j\mathbf{U}$ or $j\mathbf{U} \prec i\mathbf{U}$. Let $\mathbf{L}_\mathbf{U}$ denote the sequential program that consists of the iterations of \mathbf{L} with this new execution order. The mapping $T_\mathbf{U} : \mathbf{L} \mapsto \mathbf{L}_\mathbf{U}$ (on the set of perfect loop nests of length m) is the *unimodular transformation* defined by the matrix \mathbf{U} . The program $\mathbf{L}_\mathbf{U}$ is the *transformed program* of \mathbf{L} defined by \mathbf{U} .

The transformed program $\mathbf{L}_\mathbf{U}$ can be represented as a perfect nest of m **do** loops (like \mathbf{L}), with an index vector \mathbf{K} that is related to the index vector of \mathbf{L} by the equation $\mathbf{K} = \mathbf{IU}$. The loops in $\mathbf{L}_\mathbf{U}$ from the outermost inward will be denoted by $L_{U1}, L_{U2}, \dots, L_{Um}$. The limits of these loops are obtained from the limits of the loops of \mathbf{L} by Algorithm 3.2 (see Example 2.1). An index value \mathbf{i} in \mathbf{L}

is transformed into an index value $i\mathbf{U}$ in $\mathbf{L_U}$. The condition for equivalence of $\mathbf{L_U}$ to \mathbf{L} is that $i\mathbf{U} \prec j\mathbf{U}$ whenever $H(j)$ depends on $H(i)$ in \mathbf{L} , that is, $\mathbf{dU} \succ \mathbf{0}$ for each positive distance vector \mathbf{d} in \mathbf{L} (proof?). If $\mathbf{L_U}$ is equivalent to \mathbf{L} , then a distance vector \mathbf{d} in \mathbf{L} will become a distance vector \mathbf{dU} in $\mathbf{L_U}$, and all the distance vectors in $\mathbf{L_U}$ are accounted for in this way.

In this section, we will illustrate unimodular transformations in terms of double loops. We will show how a double loop can be transformed by a 2×2 unimodular matrix into an equivalent double loop where the outer or the inner loop can execute in parallel.

Example 6.5 Consider the double loop $\mathbf{L} = (L_1, L_2)$:

```

 $L_1 :$       do  $I_1 = 5, 100$ 
 $L_2 :$       do  $I_2 = 5, 100$ 
 $$             $X(I_1, I_2) = X(I_1, I_2 - 1) + X(I_1 - 2, I_2 + 3) +$ 
 $$             $X(I_1 - 3, I_2 + 7)$ 
 $$            enddo
 $$            enddo

```

It has three (uniform) distance vectors: $(0, 1), (2, -3), (3, -7)$. Since there is a dependence at level 1, the outer loop cannot execute in parallel (Exercise 6.1.1). Also, since there is a dependence at level 2, the inner loop cannot execute in parallel. We will find a unimodular matrix \mathbf{U} such that the transformed program $\mathbf{L_U} = (L_{U1}, L_{U2})$ is equivalent to \mathbf{L} and its inner loop L_{U2} can execute in parallel.

For $\mathbf{L_U}$ to be equivalent to \mathbf{L} , we must have $\mathbf{dU} \succ \mathbf{0}$ for each distance vector \mathbf{d} in \mathbf{L} . In order that the inner loop L_{U2} may execute in parallel, there must not be any dependence in $\mathbf{L_U}$ at level 2. Since the distance vectors in $\mathbf{L_U}$ are of the form \mathbf{dU} where \mathbf{d} is a distance vector in \mathbf{L} , this means $\mathbf{dU} \succ_2 \mathbf{0}$ is not allowed for any \mathbf{d} . Thus, for both conditions to hold, we must have $\mathbf{dU} \succ_1 \mathbf{0}$ for each distance vector \mathbf{d} in \mathbf{L} . We seek a 2×2 unimodular matrix $\mathbf{U} = (u_{rs})$ such that the first component of each \mathbf{dU} is positive, that is

$$d_1 u_{11} + d_2 u_{21} \geq 1$$

for $\mathbf{d} \in \{(0, 1), (2, -3), (3, -7)\}$. The set of three inequalities can

be written as:

$$\left. \begin{array}{l} u_{21} \geq 1 \\ 2u_{11} - 3u_{21} \geq 1 \\ 3u_{11} - 7u_{21} \geq 1 \end{array} \right\}$$

which is equivalent to (Algorithm 3.2)

$$\left. \begin{array}{l} u_{11} \geq \max\left(\frac{3}{2}u_{21}, \frac{7}{3}u_{21}\right) \\ u_{21} \geq 1. \end{array} \right\}$$

The smallest possible value of u_{21} is 1, and with this choice the smallest possible value of u_{11} is $\lceil \max(3/2, 7/3) \rceil$ or 3. Thus, one possible choice for the first column of \mathbf{U} is $(u_{11}, u_{21}) = (3, 1)$.

A unimodular matrix \mathbf{U} whose first column is $(3, 1)$ is obtained by the method of Corollary 4 to Theorem 3.4. By Algorithm 2.2, we find the (modified) echelon reduction of the matrix $\begin{pmatrix} 3 \\ 1 \end{pmatrix}$:

$$\begin{pmatrix} 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The element in the first row of the echelon matrix is already positive, so that no further changes are needed to be made in the unimodular and the echelon matrices. One unimodular matrix that has $(3, 1)$ for its first column is

$$\mathbf{U} = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix}, \text{ and its inverse is } \mathbf{U}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -3 \end{pmatrix}.$$

This matrix \mathbf{U} will define a valid unimodular transformation of \mathbf{L} such that the inner loop in the transformed program can execute in parallel.

We have to change over from the index variables I_1, I_2 to new index variables K_1, K_2 defined by $(K_1, K_2) = (I_1, I_2)\mathbf{U}$. The constraints on I_1, I_2 are

$$(5, 5) \leq (I_1, I_2) \leq (100, 100).$$

Since

$$(I_1, I_2) = (K_1, K_2)\mathbf{U}^{-1} = (K_2, K_1 - 3K_2),$$

the constraints in terms of K_1, K_2 are

$$\begin{array}{lcl} 5 & \leq & K_2 & \leq & 100 \\ 5 & \leq & K_1 - 3K_2 & \leq & 100. \end{array}$$

Using Algorithm 3.2, we get

$$\begin{array}{lcl} \lceil \max(5, (K_1 - 100)/3) \rceil & \leq & K_2 & \leq & \lfloor \min(100, (K_1 - 5)/3) \rfloor \\ 20 & \leq & K_1 & \leq & 400. \end{array}$$

The transformed program of **L** defined by **U** is then

```
 $L_{U1} : \quad \text{do } K_1 = 20, 400$ 
 $L_{U2} : \quad \text{do } K_2 = \lceil \max(5, (K_1 - 100)/3) \rceil, \lfloor \min(100, (K_1 - 5)/3) \rfloor$ 
 $\quad X(K_2, K_1 - 3K_2) = X(K_2, K_1 - 3K_2 - 1) +$ 
 $\quad \quad X(K_2 - 2, K_1 - 3K_2 + 3) +$ 
 $\quad \quad X(K_2 - 3, K_1 - 3K_2 + 7)$ 
 $\quad \text{enddo}$ 
 $\text{enddo}$ 
```

where the new body is obtained by replacing I_1 by K_2 and I_2 by $K_1 - 3K_2$ in the body of **L**. This program is equivalent to **L**. The set of distance vectors in (L_{U1}, L_{U2}) is

$$\{\mathbf{d}\mathbf{U} : \mathbf{d} = (0, 1), (2, -3), (3, -7)\} = \{(1, 0), (3, 2), (2, 3)\}.$$

Since now there is no dependence at level 2, the inner loop in the transformed program can execute in parallel, so that the K_2 -loop can be changed into a **doall** loop.

Example 6.6 Consider the loop nest **L**:

```
 $L_1 : \quad \text{do } I_1 = 5, 100$ 
 $L_2 : \quad \text{do } I_2 = 16, 80$ 
 $\quad X(I_1, I_2) = X(I_1 - 2, I_2 - 4) + X(I_1 - 3, I_2 - 6)$ 
 $\quad \text{enddo}$ 
 $\text{enddo}$ 
```

It has two (uniform) distance vectors: $(2, 4), (3, 6)$. Since there is a dependence at level 1, the outer loop cannot execute in parallel. We will find a unimodular matrix **U** such that the transformed

program $\mathbf{L_U} = (L_{U1}, L_{U2})$ is equivalent to \mathbf{L} and its outer loop L_{U1} can execute in parallel.

For $\mathbf{L_U}$ to be equivalent to \mathbf{L} , we must have $\mathbf{dU} \succ \mathbf{0}$ for each distance vector \mathbf{d} in \mathbf{L} . In order that the outer loop L_{U1} may execute in parallel, there must not be any dependence in $\mathbf{L_U}$ at level 1; that is, $\mathbf{dU} \succ_1 \mathbf{0}$ is not allowed for any \mathbf{d} . Thus, for both conditions to hold, we must have $\mathbf{dU} \succ_2 \mathbf{0}$ for each distance vector \mathbf{d} in \mathbf{L} . We seek a 2×2 unimodular matrix \mathbf{U} such that the first component of each \mathbf{dU} is zero and the second component is positive, that is

$$\begin{aligned} d_1 u_{11} + d_2 u_{21} &= 0 \\ d_1 u_{12} + d_2 u_{22} &\geq 1 \end{aligned}$$

for $\mathbf{d} \in \{(2, 4), (3, 6)\}$.

The dependence matrix of \mathbf{L} is

$$\mathcal{D} = \begin{pmatrix} 2 & 4 \\ 3 & 6 \end{pmatrix}.$$

The unimodular matrix \mathbf{U} we are searching for must be such that the first column of the product $\mathcal{D}\mathbf{U}$ is $(0, 0)$ and the second column has positive elements. Using Algorithm 2.1, we reduce the transpose of the dependence matrix to echelon form:

$$\begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 \\ 4 & 6 \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 0 & 0 \end{pmatrix}.$$

To make the zero row the first row in the echelon matrix, we pre-multiply both sides by the 2×2 interchange matrix:

$$\begin{pmatrix} -2 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 2 & 3 \\ 4 & 6 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 2 & 3 \end{pmatrix}.$$

Taking transposes of both sides, we get⁵

$$\begin{pmatrix} 2 & 4 \\ 3 & 6 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 2 \\ 0 & 3 \end{pmatrix}. \quad (6.1)$$

⁵Remember that $(\mathbf{AB})' = \mathbf{B}'\mathbf{A}'$.

The unimodular matrix

$$\mathbf{U} = \begin{pmatrix} -2 & 1 \\ 1 & 0 \end{pmatrix}$$

satisfies all our requirements. Thus, \mathbf{U} will define a valid transformation such that the outer loop L_{U1} in the transformed program can execute in parallel.

The constraints on I_1 and I_2 are

$$(5, 16) \leq (I_1, I_2) \leq (100, 80).$$

Since

$$(I_1, I_2) = (K_1, K_2)\mathbf{U}^{-1} = (K_2, K_1 + 2K_2),$$

the constraints in terms of K_1 and K_2 are

$$\begin{array}{rcl} 5 & \leq & K_2 & \leq & 100 \\ 16 & \leq & K_1 + 2K_2 & \leq & 80. \end{array}$$

Using Algorithm 3.2, we get

$$\begin{array}{rcl} \lceil \max(5, 8 - K_1/2) \rceil & \leq & K_2 & \leq & \lfloor \min(100, 40 - K_1/2) \rfloor \\ -184 & \leq & K_1 & \leq & 70. \end{array}$$

Hence, the transformed program defined by \mathbf{U} is

```

 $L_{U1} :$    do  $K_1 = -184, 70$ 
 $L_{U2} :$    do  $K_2 = \lceil \max(5, 8 - K_1/2) \rceil, \lfloor \min(100, 40 - K_1/2) \rfloor$ 
             $X(K_2, K_1 + 2K_2) = X(K_2 - 2, K_1 + 2K_2 - 4)$ 
             $\quad\quad\quad + X(K_2 - 3, K_1 + 2K_2 - 6)$ 
            enddo
enddo
```

The set of distance vectors in (L_{U1}, L_{U2}) is

$$\{\mathbf{d}\mathbf{U} : \mathbf{d} = (2, 4), (3, 6)\} = \{(0, 2), (0, 3)\}.$$

There is no dependence at level 1; the outer loop L_{U1} can execute in parallel, so that it can be changed into a **doall** loop.

The material for this section has been derived from [Bane91]. The research in this area can be traced back to Leslie Lamport [Lamp74]; in fact, we have used Lamport's Hyperplane method in Example 6.5. Work on unimodular transformations has been done by Mike Wolfe [Wolf86a], François Irigoin and Rémi Triolet [IrTr89], Michael Wolf and Monica Lam ([WoLa91a], [WoLa91b]), Michael Dowling [Dowl90], and others.

EXERCISES 6.3

- For each 2×2 elementary matrix \mathbf{U} (see Example 2.2), find a set of necessary and sufficient conditions on the distance vectors of a double loop \mathbf{L} such that the unimodular transformation of \mathbf{L} defined by \mathbf{U} is valid.
- In Example 6.5, there are infinitely many possible choices for (u_{11}, u_{21}) . Take $u_{21} = 2$ and then choose a suitable value for u_{11} . Find a unimodular matrix with this new first column, and compute the transformed program for that matrix. Compare the new transformed program with the one obtained in the example. Is one 'better' than the other in some sense?
- By the method of Example 6.5, find a unimodular transformation of the given loop nest such that the inner loop in the transformed program can execute in parallel:

```
(a)  $L_1 : \quad \text{do } I_1 = 0, 100$   

       $L_2 : \quad \text{do } I_2 = 0, 40$   

       $S : \quad X(I_1, I_2) = X(I_1 - 1, I_2) + X(I_1, I_2 - 1)$   

            enddo  

            enddo
```

```
(b)  $L_1 : \quad \text{do } I_1 = 0, 100$   

       $L_2 : \quad \text{do } I_2 = 0, I_1$   

       $S : \quad X(I_1, I_2) = X(I_1 - 1, I_2) + X(I_1, I_2 - 1) +$   

             $X(I_1 - 2, I_2 + 1)$   

            enddo  

            enddo
```

- Replace the body of the loop nest of Example 6.6 by the statement

$$X(I_1, I_2) = X(I_1 - 3, I_2 + 6) + X(I_1 - 4, I_2 + 8).$$

By the method of Example 6.6, find a unimodular transformation of the changed loop nest such that the outer loop in the transformed program can execute in parallel.

5. Prove or give a counter example:

- (a) For every double loop, there is a unimodular transformation such that the inner loop in the transformed program can execute in parallel.
- (b) For every double loop, there is a unimodular transformation such that the outer loop in the transformed program can execute in parallel.

6.4 Loop Permutations

A permutation matrix is also a unimodular matrix. The unimodular transformation of a loop nest by a permutation matrix is called a *loop permutation*. The condition for validity of a loop permutation can be expressed in terms of the direction vectors in the loop nest, while a general unimodular transformation requires distance vectors. Since direction vectors are easier to compute than distance vectors, a loop permutation is easier to handle than a general unimodular transformation.

A permutation matrix permutes the components of a distance vector \mathbf{d} in the same way as it permutes the components of the sign of \mathbf{d} , so that $\text{sig}(\mathbf{d}\mathcal{P}) = \text{sig}(\mathbf{d}) \cdot \mathcal{P}$. For example, consider the permutation matrix

$$\mathcal{P} = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}. \quad (6.2)$$

Taking a distance vector $\mathbf{d} = (2, 5, -3)$ and the corresponding direction vector $\boldsymbol{\sigma} = \text{sig}(\mathbf{d}) = (1, 1, -1)$, we see that

$$\mathbf{d}\mathcal{P} = (-3, 5, 2) \quad \text{and} \quad \boldsymbol{\sigma}\mathcal{P} = (-1, 1, 1).$$

Thus, the fact that \mathcal{P} acting on \mathbf{d} would turn \mathbf{d} into a negative vector can be deduced from the action of \mathcal{P} on $\boldsymbol{\sigma}$. In contrast, as we can see from the examples in the previous section, whether or not $\mathbf{d}\mathbf{U}$ is positive cannot be deduced from whether or not $\boldsymbol{\sigma}\mathbf{U}$ is positive for a general unimodular matrix \mathbf{U} .

The condition for the validity of a (unimodular) transformation by a unimodular matrix \mathbf{U} is that $\mathbf{d}\mathbf{U} \succ \mathbf{0}$ for each positive distance

vector \mathbf{d} in \mathbf{L} (Section 6.3). The condition for the validity of a loop permutation by a permutation matrix \mathcal{P} is that $\sigma\mathcal{P} \succ \mathbf{0}$ for each positive direction vector σ in \mathbf{L} .

A positive direction vector σ is said to *prevent* the loop permutation defined by a permutation matrix \mathcal{P} if $\sigma\mathcal{P} \prec \mathbf{0}$. To see if a particular loop permutation is valid, we must test for the possible existence of all direction vectors that prevent that permutation.

Example 6.7 The loop permutation defined by the matrix in (6.2) is prevented by the direction vectors $(0, 1, -1)$, $(1, -1, 0)$, and all vectors of the form $(1, *, -1)$. The rectangular loop nest \mathbf{L}

```

do  $I_1 = p_1, q_1$ 
  do  $I_2 = p_2, q_2$ 
    do  $I_3 = p_3, q_3$ 
       $X(I_1, I_2, I_3) = X(I_1 - 3, I_2 - 4, I_3 + 2) + 1$ 
    enddo
  enddo
enddo

```

has the direction vector $(1, 1, -1)$. Hence, its permutation by the matrix of (6.2) will not be valid; that is, the loop nest

```

do  $I_3 = p_3, q_3$ 
  do  $I_2 = p_2, q_2$ 
    do  $I_1 = p_1, q_1$ 
       $X(I_1, I_2, I_3) = X(I_1 - 3, I_2 - 4, I_3 + 2) + 1$ 
    enddo
  enddo
enddo

```

is not equivalent to \mathbf{L} . On the other hand, the loop nest

```

do  $I_2 = p_2, q_2$ 
  do  $I_3 = p_3, q_3$ 
    do  $I_1 = p_1, q_1$ 
       $X(I_1, I_2, I_3) = X(I_1 - 3, I_2 - 4, I_3 + 2) + 1$ 
    enddo
  enddo
enddo

```

is equivalent to \mathbf{L} , since the direction vector of \mathbf{L} does not prevent the loop permutation defined by the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}.$$

Loop permutation defined by an interchange matrix (which is a permutation matrix) is called *loop interchange*. Loop interchange has been studied in detail by Mike Wolfe ([Wolf86b], [Wolf89]), and by John Allen and Ken Kennedy [AlKe84]. Loop interchange and other types of loop permutations are discussed in [Bane90].

EXERCISES 6.4

1. Show that interchanging the two loops in a double loop is valid iff there is no dependence with the direction vector $(1, -1)$.
2. For each 3×3 permutation matrix \mathcal{P} , find all positive direction vectors of size 3 that prevent the loop permutation of a triple loop defined by \mathcal{P} .
3. Find all valid loop permutations of the rectangular loop nest of Example 6.7 after replacing its body by the statement:

$$X(I_1, I_2, I_3) = X(I_1 - 3, I_2 + 1, I_3 - 2) + X(I_1, I_2 - 1, I_3 + 3).$$

4. Find all positive direction vectors that prevent the loop permutation defined by the permutation matrix:

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

5. Let $1 \leq p < m$. Prove that a direction vector in the nest \mathbf{L} will prevent the interchange of the adjacent loops L_p and L_{p+1} iff it is of the form

$$(\underbrace{0, 0, \dots, 0}_{p-1}, 1, -1, *, *, \dots, *).$$

6. Prove that in a sequential nest of m loops, two adjacent loops L_p and L_{p+1} can be interchanged if there is no dependence at level p .
7. Prove that in a sequential nest of m loops, a loop L_p can be moved to the innermost position if there is no dependence at level p .

6.5 Loop Distribution

A *statement-level* loop transformation of \mathbf{L} is a transformation that treats the instances of assignment statements in \mathbf{L} as indivisible units, and changes their order of execution. *Loop Distribution* is a statement-level transformation based on the acyclic decomposition of the statement dependence graph of \mathbf{L} . (See Section 1.4.) Before giving a definition, we will illustrate the transformation by two examples:

Example 6.8 Consider the loop

```
L :      do I = 4, 100
         S1 :      A(I) = B(I - 2) + 1
         S2 :      C(I) = B(I - 1) + F(I)
         S3 :      B(I) = A(I - 1) + 2
         S4 :      D(I) = D(I - 1) + B(I - 1)
      enddo
```

The dependence relation in the set of statements of L is as follows:

$$S_1 \delta^f S_3, S_3 \delta^f S_1, S_3 \delta^f S_2, S_3 \delta^f S_4, S_4 \delta^f S_4.$$

The statement dependence graph G of L is shown in Figure 6.6(a). We find the condensation \tilde{G} of G by the Tarjan algorithm; it is shown in Figure 6.6(b).

For each strongly connected component of G , we construct a loop by deleting from L all statements not in that component. Thus, the loops corresponding to $\{S_1, S_3\}$, $\{S_2\}$ and $\{S_4\}$ are

```
L1 :      do I = 4, 100
         S1 :      A(I) = B(I - 2) + 1
         S3 :      B(I) = A(I - 1) + 2
      enddo

L2 :      do I = 4, 100
         S2 :      C(I) = B(I - 1) + F(I)
      enddo
```

and

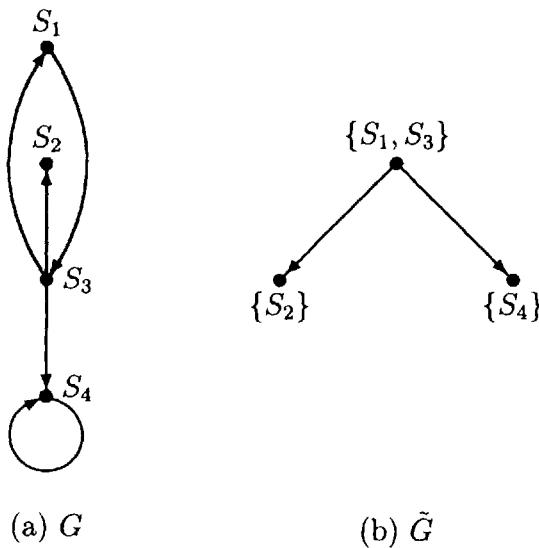


Figure 6.6: The digraph of Example 6.8 and its condensation.

$L_3 :$ **do** $I = 4, 100$
 $S_4 :$ $D(I) = D(I - 1) + B(I - 1)$
enddo

respectively. Loop distribution applied to \mathbf{L} changes the sequential execution order of the statement instances to one defined by the following rules:

1. First, execute the loop L_1 .
 2. When L_1 has finished, start executing L_2 and L_3 simultaneously.

It is clear that an instance $S_b(j)$ is executed after an instance $S_a(i)$ in the transformed program whenever $S_b(j)$ depends on $S_a(i)$ in L ($1 \leq a \leq 4$, $1 \leq b \leq 4$), since

1. The relative ordering of the instances of S_1 and S_3 is the same as in L ,
 2. Each instance of S_2 is executed after each instance of S_3 ,

3. Each instance of S_4 is executed after each instance of S_3 , and
4. The relative ordering of the instances of S_4 is the same as in L .

Therefore, the transformation is valid.

From the equivalent program described above, we can derive other programs equivalent to \mathbf{L} . Note that L_2 can be changed into a **doall** loop since there is no dependence in that loop. (This is not possible for the other two loops.) That **doall** loop with a single statement can be written in the form

$$C(4 : 100 : 1) = B(3 : 99 : 1) + F(4 : 100 : 1)$$

using ‘array sections.’ This is an example of ‘vector code.’ Also, instead of executing L_2 and L_3 simultaneously, we can execute the three loops in one of the two sequences:

$$L_1, L_2, L_3 \text{ or } L_1, L_3, L_2.$$

Example 6.9 Consider the double loop of Example 4.3:

```

 $L_1 :$       do  $I_1 = 0, 4$ 
 $L_2 :$       do  $I_2 = 0, 4$ 
 $S :$          $A(I_1 + 1, I_2) = B(I_1, I_2) + C(I_1, I_2)$ 
 $T :$          $B(I_1, I_2 + 1) = A(I_1, I_2 + 1) + 1$ 
 $U :$          $D(I_1, I_2) = B(I_1, I_2 + 1) - 2$ 
          enddo
          enddo

```

The graph of Figure 6.7(a) is the statement dependence graph of (L_1, L_2) . It has two components $\{S, T\}$ and $\{U\}$, and the first component precedes the second in its condensation (not shown). The subprograms corresponding to $\{S, T\}$ and $\{U\}$ are

```

 $L_{11} :$       do  $I_1 = 0, 4$ 
 $L_{12} :$       do  $I_2 = 0, 4$ 
 $S :$          $A(I_1 + 1, I_2) = B(I_1, I_2) + C(I_1, I_2)$ 
 $T :$          $B(I_1, I_2 + 1) = A(I_1, I_2 + 1) + 1$ 
          enddo
          enddo

```

and

```

 $L_{21} :$       do  $I_1 = 0, 4$ 
 $L_{22} :$       do  $I_2 = 0, 4$ 
 $U :$            $D(I_1, I_2) = B(I_1, I_2 + 1) - 2$ 
              enddo
              enddo

```

respectively. After loop distribution is applied to (L_1, L_2) , we get the program where (L_{11}, L_{12}) is executed first and then (L_{21}, L_{22}) . Note that each loop of (L_{21}, L_{22}) can be changed into a **doall** loop since there is no dependence at any level in that nest.

Figure 6.7(b) shows the dependence graph of an instance of loop L_2 for a fixed value of I_1 . The subgraph consisting of S and T is the dependence graph of an instance of L_{12} for a fixed value of I_1 . (The edge from S to T has disappeared.) Loop distribution applied to an instance of L_{12} will yield the following program:

```

 $L_{121} :$       do  $I_2 = 0, 4$ 
 $T :$            $B(I_1, I_2 + 1) = A(I_1, I_2 + 1) + 1$ 
              enddo
 $L_{122} :$       do  $I_2 = 0, 4$ 
 $S :$            $A(I_1 + 1, I_2) = B(I_1, I_2) + C(I_1, I_2)$ 
              enddo

```

(We get two loops L_{121} and L_{122} , and L_{122} must be executed after L_{121} .) Note that each of the two loops can be changed into a **doall** loop. Thus, (L_{11}, L_{12}) is equivalent to the program

```

do  $I_1 = 0, 4$ 
    doall  $I_2 = 0, 4$ 
         $B(I_1, I_2 + 1) = A(I_1, I_2 + 1) + 1$ 
    enddoall
    doall  $I_2 = 0, 4$ 
         $A(I_1 + 1, I_2) = B(I_1, I_2) + C(I_1, I_2)$ 
    enddoall
enddo

```

and the original loop nest (L_1, L_2) is equivalent to the program

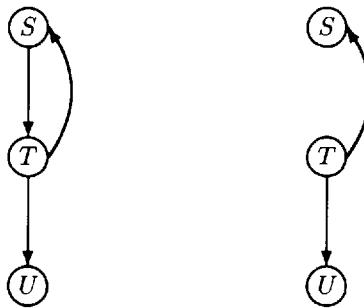
(a) Levels ≥ 1 (b) Levels ≥ 2

Figure 6.7: Statement dependence graphs for Example 6.9.

```

do  $I_1 = 0, 4$ 
    doall  $I_2 = 0, 4$ 
     $T : \quad B(I_1, I_2 + 1) = A(I_1, I_2 + 1) + 1$ 
        enddoall
        doall  $I_2 = 0, 4$ 
         $S : \quad A(I_1 + 1, I_2) = B(I_1, I_2) + C(I_1, I_2)$ 
        enddoall
    enddo
    doall  $I_1 = 0, 4$ 
        doall  $I_2 = 0, 4$ 
         $U : \quad D(I_1, I_2) = B(I_1, I_2 + 1) - 2$ 
        enddoall
    enddoall

```

Consider now a general sequential loop nest \mathbf{L} . Let G denote its statement dependence graph, \tilde{G} the acyclic condensation of G , and \leq the partial order in the set of strongly connected components of G (Chapter 1). Arrange those components in a hierarchy of maximal antichains (S_1, S_2, \dots, S_n) such that

1. If $i < j$, then no component in S_i has a predecessor in S_j ;
2. For $1 < i \leq n$, each component in S_i has at least one immediate predecessor in S_{i-1} .

(See Theorem 1.1.) For each strongly connected component C of G , define a loop nest that is obtained from \mathbf{L} by deleting all statements not in C . *Loop Distribution* is a valid transformation that when applied to \mathbf{L} will change its sequential execution order into one defined by the following rules:

1. Start executing simultaneously the loop nests for all the components in S_1 ;
2. For $1 < i \leq n$, start the loop nest for a component C in S_i as soon as the loop nests for all the immediate predecessors of C in S_{i-1} have finished.

Several variations on this scheme are possible leading to equivalent programs. For example, if a strongly connected component C consists of a single statement that does not depend on itself, then the all the loops in the loop nest for C can be changed into **doall** loops. The partially ordered set of strongly connected components can be topologically sorted, and the loop nests corresponding to the components can then be executed in a sequence in the same order.

Loop distribution was introduced by Yoichi Muraoka [Mura71]. It is discussed by Banerjee et al. [BCKT79], Ken Kennedy and Kathryn McKinley [KeMc90], and others.

An introduction to loop transformations is given in [PaWo86]; for a more comprehensive and up-to-date account, see [BENP93].

For any dependence, we can derive the direction vectors if the distance vectors are known, and derive the levels if the direction vectors are known. However, it does not work in the opposite direction: a direction vector may come from many distance vectors and a level may come from many direction vectors. In general, direction vectors are easier to compute than distance vectors and levels are easier than direction vectors. As we have seen in this chapter, different amounts of dependence information are needed for different loop transformations: distance vectors for iteration graph partitioning and general unimodular transformations, direction vectors for loop permutations, and levels for loop distribution.

EXERCISES 6.5

Apply loop distribution (repeatedly, if possible) to get a program equivalent to the given loop nest. Point out which **do** loops can be changed into **doall** loops.

1. $L :$ **do** $I = 4, 200$

$$\begin{array}{ll} S : & A(I) = B(I) + C(I) \\ T : & B(I+2) = A(I-1) + C(I-1) \\ U : & A(I+1) = B(2I+3) + 1 \end{array}$$

enddo

2. $L :$ **do** $I = 5, 200$

$$\begin{array}{ll} S_1 : & A(I-1) = E(3I) + 2 \\ S_2 : & B(I) = A(I) + C(I-1) + C(I-3) \\ S_3 : & D(2I) = B(I-2) + 1 \\ S_4 : & C(I-2) = D(2I+1) - 1 \\ S_5 : & B(I+1) = E(3I+1) + 24 \end{array}$$

enddo

3. $L_1 :$ **do** $I_1 = 20, 200$

$L_2 :$ **do** $I_2 = 10, I_1 + 400$

$$\begin{array}{ll} S : & A(I_1, I_2) = B(I_1, I_2) + C(I_1 + 1, I_2) \\ T : & C(I_1, I_2) = A(I_1 - 1, I_2) + 1 \\ U : & D(I_1, I_2) = C(I_1 - 2, I_2) - 2 \end{array}$$

enddo

enddo

4. $L_1 :$ **do** $I_1 = 20, 200$

$L_2 :$ **do** $I_2 = 10, I_1 + 400$

$$\begin{array}{ll} S : & A(I_1, I_2) = B(I_1, I_2) + C(I_1, I_2 + 1) \\ T : & C(I_1, I_2) = A(I_1, I_2 - 1) + 1 \\ U : & D(I_1, I_2) = C(I_1, I_2 - 2) - 2 \end{array}$$

enddo

enddo

Appendix

Code Examples

The Appendix contains sample implementations (in ANSI C) of the major algorithms in the text taken in the following order:

1. Echelon Reduction (Algorithm 2.1)
2. Diophantine Equations 1 (Theorem 3.6)
3. Diagonalization (Algorithm 2.3)
4. Diophantine Equations 2 (Theorem 3.7)
5. Two-Variable Equation under Constraints (Algorithm 3.1)
6. Fourier Elimination (Algorithm 3.2)
7. Linear Dependence (Algorithms 5.1, 5.2, 5.3).

Whenever necessary, an algorithm may utilize any previous algorithm in this sequence. Each algorithm has a driver associated with it and the core routines are invoked from the driver. A driver routine takes input from the terminal and prints the output. The core routines can be incorporated into a compiler with minor modifications for the compiler's data structures. Some utility routines for rational arithmetic needed by Fourier Elimination are also included. The code for each algorithm is followed by some sample inputs and outputs (with some simple formatting changes). All the coding was done by K. Sridharan of Intel Corporation.

Basic Header Files

```
/*
***      File  style.h
*/
#define           STYLE_INCLUDED
/*
***      Defines to make key words uppercase to make
***      code more readable.
*/
#define  INT      int
#define  DOUBLE   double
#define  CHAR     char
#define  STRUCT   struct
#define  TYPEDEF  typedef
#define  FOR      for
#define  VOID     void
#define  WHILE    while
#define  IF       if
#define  ELSE     else

#define  BREAK    break
#define  EXTERN   extern
#define  RETURN   return
/*
***      Some basic macros for general use.
*/
#define  SIG(num)      (num > 0 ? 1 : (num<0 ? -1:0))
#define  ABS(num)      (num < 0 ? -num : num)
#define  MAX(a,b)      (a>b?a:b)
#define  MIN(a,b)      (a<b?a:b)
#define  PLUS_PART(num) (num >= 0 ? num : 0 )
#define  MINUS_PART(num) (num < 0 ? -num : 0 )
#define  TRUE          1
#define  FALSE         0
/*
***      Some basic typedefs
*/
TYPEDEF  INT      BOOL;
```

```
***  
***   File  "two_var.h"  
***  
**/  
#define TWO_VAR_INCLUDED  
#ifndef STYLE_INCLUDED  
#include "style.h"  
#endif  
  
TYPEDEF STRUCT {  
    INT    a_part;  
    INT    b_part;  
) PAIRS;  
  
TYPEDEF STRUCT {  
    BOOL   empty_set;  
    PAIRS x;  
    PAIRS y;  
    INT    lbnd_t;  
  
    INT    ubnd_t;  
) Psi;  
  
TYPEDEF      STRUCT {  
    INT    num;  
    INT    denom;  
) RATIONAL_NODE;  
  
TYPEDEF RATIONAL_NODE      *RATIONAL;  
EXTERN RATIONAL rat_div(RATIONAL a, RATIONAL b);  
EXTERN RATIONAL rat_mul(RATIONAL a, RATIONAL b);  
EXTERN RATIONAL rat_add(RATIONAL a, RATIONAL b);  
EXTERN RATIONAL rat_sub(RATIONAL a, RATIONAL b);  
EXTERN RATIONAL rat_max(RATIONAL a, RATIONAL b);  
EXTERN RATIONAL rat_min(RATIONAL a, RATIONAL b);  
EXTERN BOOL      rat_eqt(RATIONAL a, RATIONAL b);  
EXTERN BOOL      rat_grt(RATIONAL a, RATIONAL b);  
EXTERN BOOL      rat_lst(RATIONAL a, RATIONAL b);
```

Driver For Echelon Reduction

```

/*
***  File "new_ech.c"
*/
#include <stdlib.h>
#include <stdio.h>
#include "style.h"
/***
****  New_ech.c:  A driver for calling the echelon
****      reduction algorithm.
**** 
****  Reads input from terminal for values of m, n,
****  allocates storage for matrix A, reads values
****  for A, calls the echelon reduction algorithm
****  and writes output matrices U and S to a file.
****  This reads values interactively.
/**/
EXTERN VOID echelon_reduc ( INT *, INT *, INT *, INT, INT);

main() {
    INT i, j, k, m, n;
    INT *A, INT * U, INT * S;
    CHAR fname[FILENAME_MAX];
    FILE * file_ptr;

    /**
    ***  Read the output file name, and open the file
    ***  for output.
    **/ 
    printf("Enter file name for output:\n");
    scanf("%s", fname);
    file_ptr = fopen(fname, "a");

    /**
    ***  Read the values of m and n.
    ***  Allocate storage for A, U and S.
    **/ 

    printf("Enter the Number m:\n");
    scanf("%d", &m);
    printf("The Number m is : %d\n",m);
    printf("Enter the Number n:\n");
    scanf("%d", &n);
    printf("The Number n is: %d\n",n);
    A = (INT *)calloc(m*n,sizeof(int));
}

```

```

S = (INT *)calloc(m*n,sizeof(int));
U = (INT *)calloc(m*m,sizeof(int));
/***
*** Read the matrix A and write it out to
*** the output file.
***/
FOR(i = 0; i < m; i++) {
    printf("Enter Row %d of matrix A:\n",i);
    FOR(j = 0; j < n; j++) {
        scanf("%d",A+i*n+j);
    }
}
fprintf(file_ptr,
"\n THE MATRIX A Before Ech. Transform. is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR (j = 0; j < n; j++)
        fprintf(file_ptr, " %4d ", A[i*n + j]);
    fprintf(file_ptr, "\n");
}

/***
*** Call the echelon reduction algorithm
***/
echelon_reduc (A, U, S, m, n);
/***
*** Output the result matrices U and S to the file.
***/
fprintf(file_ptr,
"\n THE MATRIX U After Ech. Transform. is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR (j = 0; j < m; j++)
        fprintf(file_ptr, " %4d ", U[i*m + j]);
    fprintf(file_ptr, "\n");
}
fprintf(file_ptr,
"\n THE MATRIX S After Ech. Transform. is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR(j = 0; j < n; j++) {
        fprintf(file_ptr, " %4d ",S[i*n + j]);
    }
    fprintf(file_ptr, "\n");
}
fclose(file_ptr);
}

```

Echelon Reduction Algorithm

```
/*
***      File "ech_red.c"
***
*/
#include "style.h"
/***
*****
*****      Echelon Reduction Routine:
*****
*****      This routine implements the Algorithm 2.1 from the
*****      book.
*****      Description: It reduces a given integer matrix A
*****              to echelon form.
*****
*****      Input:          (An m X n matrix A.)
*****
*****                  Pointer to the integer matrix A,
*****
*****                  Pointer to the storage for U,
*****                  Pointer to the storage for S,
*****                  integer m,    and
*****                  integer n.
*****
*****      Output:         An m X m unimodular integer matrix U,
*****                  An m X n echelon integer matrix S,
*****                  such that UA = S.
*****
*****      The algorithm follows the steps in the book very
*****      closely.
*****
*****      The only requirement is that storage for U and S is
*****      pre-allocated.
***/
```

```
VOID echelon_reduc
{
    INT      *A,      /** Input integer matrix A **/
    INT      *U,      /** Output integer matrix U **/
    INT      *S,      /** Output integer echelon matrix S  **/
    INT      m,       /** integer m (the row size for A)   **/
    INT      n       /** integer n (the column size for A)**/
}
```

```

INT          i, j, k, 10, 11, q, sig_s, temp;
/***
***      Setting   U = Im (m X m identity matrix)
***      and       S = A
**/
FOR(1 = 0; i < m; i++) {
    FOR(j = 0; j < m; j++)
        U[i*m + j] = 0;
    U[i*m + i] = 1;

    FOR(j = 0; j < n; j++)
        S[i*n + j] = A[i*n + j];
}

/***
***      echelon reduction:
**/
10 = -1;

FOR(j = 0; j < n; j++) {
    FOR(11 = 10+1; 11 < m; 11++) {
        IF(S[11*n + j]) {
            10++;
            FOR(i = m-1; i > 10; i--)

                WHILE(S[i*n + j]){
                    sig_s = SIG(S[(i-1)*n + j] * S[i*n + j]);
                    q = ABS(S[(i-1)*n + j])/ABS(S[i*n + j]);
                    IF(q){
                        FOR(k=0; k < m; k++)
                            U[(i-1)*m + k] -=
                                sig_s * q * U[i*m + k];
                        FOR(k=0; k < n; k++)
                            S[(i-1)*n + k] -=
                                sig_s * q * S[i*n + k];
                    }
                    FOR(k = 0; k < m; k++) {
                        temp = U [(i-1)*m + k];
                        U[(i-1)*m + k] = U[i*m + k];
                        U[i*m + k] = temp;
                    }
                }
        }
    }
}

```

```
FOR(k =0; k < n; k++) {
    temp = S [(i-1)*n + k];
    S[(i-1)*n + k] = S[i*n + k];
    S[i*n + k] = temp;
}
}
BREAK;
}
}
}
```

Output From a Sample Run (Algorithm 2.1)

ORIG.	MATRIX A			RESULT MAT.			U	ECHELON MAT.			S
(1)	4	4	1	0	1	-1		2	-3	-1	
	6	0	1	1	0	-1		0	1	-1	
	4	3	2	-9	-2	12		0	0	13	
(2)	1	0	0	0	0	1		1	-1	1	
	2	2	-1	1	0	-1		0	1	-1	
	1	-1	1	-4	1	2		0	0	1	
(3)	5	4	-2	1	-2	2		1	-10	8	
	-10	-13	8	18	21	-10		0	-1	2	
	-12	-20	13	44	52	-25		0	0	3	
(4)	1	2	2	1	0	0		1	2	2	
	-6	8	5	0	1	1		0	2	2	
	6	-6	-3	-6	9	10		0	0	3	
(5)	3	1	-2	4	0	1	0	1	0	2	3
	1	0	2	3	0	-2	1	0	1	-5	-5
	2	1	-1	1	1	-1	-1	0	0	-3	0
(6)	1	-2	3	-1	0	-1	1	1	2	0	1
	2	-1	2	2	-1	2	-1	0	-1	-1	2
	3	1	2	3	5	-7	3	0	0	7	-10

Driver For Diophantine Equation Solver (1)

```

/*
***      File "new_dio.c"
***
*/
#include <stdlib.h>
#include <stdio.h>
#include "style.h"

EXTERN VOID diophantine_1 ( INT *, INT *, INT *, INT *,
                           INT *, int, int, INT *);

/*
****      Driver for diophantine routine 1:
****      Will read A and C, print out a message
****      indicating if there is a solution, and
****      will print the solution if there is one.

*/
main() {
    INT      i,j,k,m,n, temp_sum;
    INT      *A; INT * U; INT * S; INT *C; INT *T;
    CHAR     fname[FILENAME_MAX];
    FILE    *file_ptr;
    INT      exist;

    printf("Enter file name for output:\n");
    scanf("%s", fname);
    file_ptr = fopen(fname, "a");
    printf("Enter the Number m:\n");
    scanf("%d", &m);
    printf("The Number m is : %d\n",m);
    printf("Enter the Number n:\n");
    scanf("%d", &n);
    printf("The Number n is: %d\n",n);
    A = (INT *)calloc(m*n,sizeof(int) );
    S = (INT *)calloc(m*n,sizeof(int) );
    U = (INT *)calloc(m*m,sizeof(int) );
    T = (INT *)calloc(m ,sizeof(int) );
}

```

```

FOR(i = 0; i < m; i++) {
    printf("Enter Row %d of matrix A\n",i);
    FOR(j = 0; j < n; j++)
        scanf("%d",A+i*n+j);
}

C = (INT *)calloc(n,sizeof(int) );

printf ("Enter the vector C\n");
FOR(j = 0; j < n; j++)
    scanf("%d",C+j);

fprintf(file_ptr,
"\n THE MATRIX A Before Echelon Transform. is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR(j = 0; j < n; j++)
        fprintf(file_ptr, " %4d ", A[i*n + j]);
    fprintf(file_ptr, "\n");
}

}

diophantine_1 (A, U, S, C, T, m, n, &exist);

if(exist < 0)
    fprintf(file_ptr,
    "No solutions exist to the set of equations\n");
else{
    fprintf(file_ptr,
    "\nSolutions to the set of equations:\n\n");

    FOR(i = 0; i < exist; i++)
        fprintf(file_ptr, " t%ld is : %4d\n", i+1,T[i]);
    fprintf(file_ptr,
        "\nThe General Solution is:\n\n");
    fprintf(file_ptr, " X = ( ");

    FOR(j = 0; j < m; j++) {
        temp_sum = 0;
        FOR (i = 0; i < exist; i++)
            temp_sum += T[i] * U[i*m + j];
        fprintf(file_ptr, " %d ", temp_sum);
        FOR(i = exist; i < m; i++) {
            if(U[i*m + j])
                fprintf(file_ptr, "+ %d * t%ld",U[i*m + j],i+1);
        }
    }
}

```

```
        }
        if(j < m-1)
            fprintf(file_ptr, " ");
    }
    fprintf(file_ptr, " )\n");
}

fprintf(file_ptr,
"\n THE MATRIX U After Echelon Transform is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR (j = 0; j < m; j++)
        fprintf(file_ptr, "%4d ", U[ i * m + j]);
    fprintf(file_ptr, "\n");
}
fprintf(file_ptr,
"\n THE MATRIX S After Echelon Transform is :\n\n");
FOR(i =0; i < m; i++) {

    FOR(j = 0; j < n; j++) {
        fprintf(file_ptr, "%4d ",S[i*n + j]);
    }
    fprintf(file_ptr, "\n");
}
fclose(file_ptr);
}
```

Diophantine Equation Solver (1)

```
/*
*** File "dio_1.c"
*/
#include "style.h"
/***
*****
***** dio_1.c:
*****
***** Diophantine Equation Routine 1.
*****
***** This implements Theorem 3.6 in the book.
*****
***** Description: Using Algorithm 2.1, it
***** decides if a given system of linear
***** diophantine equations
*****  
  

***** XA = C
*****
***** has a solution, and finds the general
***** solution if solutions exist.
*****
***** Input: (An m X n matrix A, & an n-vector C.)
*****
***** Pointer to the integer matrix A,
***** Pointer to the storage for result matrix U,
***** Pointer to the storage for result matrix S,
***** Pointer to the integer vector C,
***** Pointer to the storage for m-vector T,
***** The integer m,
***** The integer n,
***** Address of integer 'exist' for returning the
***** existence of a solution.
*****
***** Output: The matrices U and S as returned from
***** the echelon reduction algorithm, a flag to
***** indicate the existence of a solution to the
***** equation, and a general solution to be
***** derived from T.
*****
***** Steps:
```

```
*****
1. The echelon reduction routine is called to find
an m X m unimodular (integer) matrix U and an
m X n echelon (integer) matrix S such that
UA = S.

*****
The value 'rho' is then found, (0 <= rho <= m)
such that the first 'rho' rows of S are nonzero
rows and the remaining m - rho rows are zero
rows.

*****
Note: rho = rank(A) = rank(S).

*****
2. Decide if the set of equations
TS = C
has an integer solution. This is easy, since
S is an echelon matrix. rho components of T
will have definite values if there is a
solution.

*****
3. If the system has no solutions, the flag
exist will have a negative value; else, it
will have a proper value based on rho.

*****
4. The general solution to XA = C is found,
which is X = TU, where T is the general
solution to TS = C.

****/
```

```
EXTERN VOID echelon_reduc( INT *, INT *, INT *, INT, INT);

VOID diophantine_1
(
    INT      *A,
    INT      *U,
    INT      *S,
    INT      *C,
    INT      *T,
    INT      m,
    INT      n,
    INT      *exist
)
{
    INT      i, j, k, i0, l1, q, sig_s, temp;
```

```
INT          rho, a, found, nr;

< /**
***      Call echelon reduction algorithm
**/

echelon_reduc (A, U, S, m, n);

< /**
***      Find rho: Number of nonzero rows from the
***              top of S.
**/

found = 0;
FOR(i=m-1; i >= 0; i --){
    FOR(j = 0; j < n; j++)
        IF(S[i*n + j]) {
            rho = i + 1;
            found++;

            BREAK;
        }
    IF(found) BREAK;
}
< /**
***      Solve for TS = C
**/


*exist = rho;

< /**
***      Find the leading element: 11
**/


FOR(j = 0; j < n; j++)
    IF(S[j]) {
        11 = j;
        BREAK;
    }

nr = 0;

FOR(j = 11; j < n; j++) {
```

```
a = 0;
FOR(i = 0; i < nr; i++)
    a += T[i] * S[i*n + j];
IF(S[nr*n + j]) {
    T[nr] = (C[j-1]-a) / S[nr*n + j];
    IF(((C[j-1]-a) % S[nr*n + j]) != 0) {
        *exist = -1;
        RETURN;
    }
    nr++;
}
ELSE{
    IF(!(C[j-1]-a)) {
        *exist = -1;
        RETURN;
    }
}
}
```

Output From a Sample Run (Theorem 3.6)

(1)

MAT. A	VECT C	RESULT MAT. U	ECHELON MAT. S
4	8	0 1 -1	2
6		1 -2 2	0
4		0 -2 3	0

Solution to the set of equations TS = C is

$$t_1 = 4$$

The General Solution to XA = C is

$$X = (t_2, 4 + -2*t_2 + -2*t_3, -4 + 2*t_2 + 3*t_3)$$

(2)

2	1	0	4	0	0	0	1	-2	5	-1
6	3	-2	2	0	-1	2	1	0	2	7
4	0	3	8	-3	1	0	0	0	0	-2
-2	5	-1		17	-4	-3	-1	0	0	0

Solution to the set of equations TS = C is

$$t_1 = -2$$

$$t_2 = 6$$

$$t_3 = 18$$

The General Solution to XA = C is

$$X = (-54 + 17*t_4, 12 + -4*t_4, 12 + -3*t_4, 4 + -1*t_4)$$

(3)

3	6	0	1	-3
-3		1	1	0

Solution to the set of equations TS = C is

$$t_1 = -2$$

The General Solution to XA = C is

$$X = (t_2, -2 + 1*t_2)$$

(4)

MAT. A	VECT C	RESULT MAT. U	ECHELON MAT. S
10	15	3 -2	2
14		-7 5	0

No solutions exist to the set of equations

(5)

55	17	0 6 13	-1
-89		1 330 715	0
41		0 41 89	0

Solution to the set of equations TS = C is

$$t_1 = -17$$

The General Solution to XA = C is

$$X = (t_2, -102 + 330*t_2 + 41*t_3, -221 + 715*t_2 + 89*t_3)$$

(6)

14	-42	0 0 1 1	-7
21		1 0 2 2	0
-35		0 1 3 3	0
28		0 0 4 5	0

Solution to the set of equations TS = C is

$$t_1 = 6$$

The General Solution to XA = C is

$$X = (t_2, t_3, 6 + 2*t_2 + 3*t_3 + 4*t_4, 6 + 2*t_2 + 3*t_3 + 5*t_4)$$

(7)

3	10	6	0	1	-3	14
-3	14	15	1	1	0	24

No solutions exist to the set of equations

Driver For Diagonalization Routine

```
/*
***      File "new_dia.c"
***
*/
#include <stdlib.h>
#include <stdio.h>
#include "style.h"

*****
*****
*****
*****   driver for diagonalization routine.
*****
*****   Reads matrix A and calls diagonalization routine.
*****   Prints the output matrices U, D and V.
*****
****/
```



```
EXTERN VOID diag ( INT *, INT *, INT *, INT *, int, int);

main() {

    INT i,j,k,m,n;
    INT *A; INT * U; INT * D; INT *V;
    CHAR fname[FILENAME_MAX];
    FILE * file_ptr;

    printf("Enter file name for output:\n");
    scanf("%s", fname);
    file_ptr = fopen(fname, "a");
    printf("Enter the Number m:\n");
    scanf("%d", &m);
    printf("The Number m is : %d\n",m);
    printf("Enter the Number n:\n");
    scanf("%d", &n);
    printf("The Number n is: %d\n",n);
    A = (INT *)calloc(m*n,sizeof(int) );
    D = (INT *)calloc(m*n,sizeof(int) );
    U = (INT *)calloc(m*m,sizeof(int) );
```

```

V = (INT *)calloc(n*n,sizeof(int));

FOR(i = 0; i < m; i++) {
    printf("Enter Row %d of matrix A:\n",i);
    FOR(j = 0; j < n; j++) {
        scanf("%d",A+i*n+j);
    }
}

fprintf(file_ptr,
"\n THE MATRIX A Before Diagonalization is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR (j = 0; j < n; j++)
        fprintf(file_ptr, " %4d ", A[ i * n + j]);
    fprintf(file_ptr, "\n");
}

diag (A, U, V, D, m, n);

}

fprintf(file_ptr,
"\n THE MATRIX U After Diagonalization is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR (j = 0; j < m; j++)
        fprintf(file_ptr, " %4d ", U[ i * m + j]);
    fprintf(file_ptr, "\n");
}

fprintf(file_ptr,
"\n THE MATRIX D After Diagonalization is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR(j = 0; j < n; j++) {
        fprintf(file_ptr, " %4d ",D[i*n+j]);
    }
    fprintf(file_ptr, "\n");
}

fprintf(file_ptr,
"\n THE MATRIX V After Diagonalization is :\n\n");
FOR(i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        fprintf(file_ptr, " %4d ", V[ i * n + j]);
    fprintf(file_ptr, "\n");
}

fclose(file_ptr);
}

```

Diagonalization Routine

```
/*
*** File "diag.c"
***
*/
#include <values.h>
#include "style.h"

*****
***** Description: It reduces a given integer matrix A
***** to diagonal form.
*****
***** Input: An m X n integer matrix A.
*****
***** Output: An m X m unimodular matrix U

*****
***** An n X n unimodular matrix V
***** An m X n diagonal (integer) matrix D
***** such that UAV = D.
*****
***** Steps: Follows the book very closely.
*****
*/
VOID diag
{
    INT    *A,
    INT    *U,
    INT    *V,
    INT    *D,
    INT    m,
    INT    n
}
{
    INT    i, j, k, io, ii, p, q, sig_s, temp, done, found;

    /**
    *** STEP 1
    ***

```

```

***      Setting   U = Im (m X m identity matrix)
***          V = In (n X n identity matrix)
***          and   D = A
***/
***/

FOR(i = 0; i < m; i++) {
    FOR(j = 0; j < m; j++)
        U[i*m + j] = 0;
    U[i*m + i] = 1;

    FOR(j = 0; j < n; j++)
        D[i*n + j] = A[i*n + j];
}

FOR (i = 0; i < n; i++) {
    FOR (j = 0; j < n; j++)
        V[i*n + j] = 0;
    V[i*n + i] = 1;

}

/***
***      Diagonalization
**/


k = 0;
done = 0;

/***
***      STEP 2:
***      Find smallest nonzero absolute element.
**/


WHILE(!done) {
    i0 = MAXINT;
    p = q = -1;
    FOR(i = k; i < m; i++)
        FOR (j = k; j < n; j++)
            IF(ABS(D[i*n + j]) < i0 &&
                ABS(D[i*n + j]) != 0 ){
                i0 = ABS(D[i*n + j]);
                p = i;
                q = j;
            }
}

```

```

    IF( p < 0 ) {
        RETURN;
    }

    /**
     ***      STEP 3:
     ***
     ***      Interchange rows of D and columns of V
     **/


    FOR(j = 0; j < n; j++) {

        temp = D[p*n + j];
        D[p*n + j] = D[k*n + j];
        D[k*n + j] = temp;

        temp = V[j*n + q];
        V[j*n + q] = V[j*n + k];
        V[j*n + k] = temp;

    }

    /**
     ***      Interchange rows of U and columns of D
     **/


    FOR(j = 0; j < m; j++) {

        temp = D[j*n + q];
        D[j*n + q] = D[j*n + k];
        D[j*n + k] = temp;

        temp = U[p*m + j];
        U[p*m + j] = U[k*m + j];
        U[k*m + j] = temp;
    }

    /**
     ***      STEP 4
     **/


    FOR(i = k+1; i < m; i++) {

```

```

    sig_s = SIG (D[i*n + k] * D[k*n + k]);
    q     = ABS (D[i*n + k]) / ABS(D[k*n + k]);
    FOR(j=0; j < m; j++)
        U[i*m + j] -= sig_s * q * U[k*m + j];
    FOR(j=0; j < n; j++)
        D[i*n + j] -= sig_s * q * D[k*n + j];
}

FOR(j = k+1; j < n; j++) {

    sig_s = SIG (D[k*n + j] * D[k*n + k]);
    q     = ABS (D[k*n + j]) / ABS(D[k*n + k]);

    FOR(i=0; i < n; i++)
        V[i*n + j] -= sig_s * q * V[i*n + k];
    FOR(i=0; i < m; i++)
        D[i*n + j] -= sig_s * q * D[i*n + k];
}

/*
***      STEP 5 and STEP 6
*/
found = 0;

FOR(j = k+1; j < n; j++) {
    IF(ABS(D[k*n + j])) {
        found = 1;
        BREAK;
    }
}
IF(!found) FOR(i = k+1; i < m; i++) {
    IF(ABS(D[i*n + k])) {
        found = 1;
        BREAK;
    }
}
IF (!found) {
    k++;
    IF((k+1) > MIN(m,n)) {
        done = 1;
        BREAK;
    }
}
}
}
}

```

Output From A Sample Run Of Diagonalization Routine

MAT. A	RESULT MAT. U	RESULT MAT. D	RESULT MAT. V
(1)			
6 5 4 2 10 -3	1 -2 0 3 -6 1 -8 17 -2	1 0 0 4 0 0	0 1 1 2
(2)			
1 0 0 2 2 -1 1 -1 1	1 0 0 -2 1 0 -3 1 1	1 0 0 0 -1 0 0 0 1	1 0 0 0 0 1 0 1 2
(3)			
5 4 -2 -10 -13 8 -12 -20 13	1 0 0 -2 -2 1 -2 5 -2	1 0 0 0 1 0 0 0 3	1 2 0 0 -6 1 2 -7 2
(4)			
1 2 2 -6 8 5 6 -6 -3	1 0 0 -6 7 8 -12 15 17	1 0 0 0 -1 0 0 0 -6	1 -2 6 0 0 1 0 1 -4
(5)			
3 1 -2 4 1 0 2 3 2 1 -1 1	1 0 0 0 1 0 -1 1 1	1 0 0 0 0 1 0 0 0 0 3 0	0 1 -2 -3 1 -3 8 5 0 0 1 0 0 0 0 1
(6)			
1 -2 3 -1 2 -1 2 2 3 1 2 3	1 0 0 -2 1 0 -3 0 1	1 0 0 0 0 -1 0 0 0 0 -1 0	1 -1 -2 -15 0 1 0 4 0 1 1 10 0 0 1 7

Driver For Diophantine Equation Solver (2)

```


/**
*** File "new_dio2.c"
*/
#include <stdlib.h>
#include <stdio.h>
#include "style.h"

EXTERN VOID diophantine_2 ( INT *, INT *, INT *, INT *, INT *,
                           INT *, int, int, INT *);

main() {
    INT i, j, k, m, n, temp_sum;
    INT *A; INT * U; INT * D; INT *C; INT *T; INT *V;
    char fname[FILENAME_MAX];

    FILE * file_ptr;
    int exist;

    printf("Enter file name for output:\n");
    scanf("%s", fname);
    file_ptr = fopen(fname, "a");
    printf("Enter the Number m:\n");
    scanf("%d", &m);
    printf("The Number m is : %d\n",m);
    printf("Enter the Number n:\n");
    scanf("%d", &n);
    printf("The Number n is: %d\n",n);
    A = (INT *)calloc(m*n,sizeof(int) );
    D = (INT *)calloc(m*n,sizeof(int) );
    U = (INT *)calloc(m*m,sizeof(int) );
    V = (INT *)calloc(n*n,sizeof(int) );
    T = (INT *)calloc(m ,sizeof(int) );

    FOR(i = 0; i < m; i++) {
        printf("Enter Row %d of matrix A\n",i);
        FOR(j = 0; j < n; j++)
            scanf("%d",A+i*n+j);
    }
}


```

```
C = (INT *)calloc(n,sizeof(int));

printf ("Enter the vector C\n");
FOR(j = 0; j < n; j++)
    scanf("%d",C+j);

fprintf(file_ptr,
        "\n THE MATRIX A Before Diagonalization is :\n\n");
FOR(i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
        fprintf(file_ptr, " %4d ", A[ i * n + j]);
    fprintf(file_ptr, "\n");

}

diophantine_2 (A, U, V, D, C, T, m, n, &exist);

IF(exist < 0)

    fprintf(file_ptr,
            "No solutions exist to the set of equations\n");

else {

    fprintf(file_ptr,
            "\nSolutions to the set of equations:\n\n");

    FOR(i = 0; i < exist; i++)
        fprintf(file_ptr, " t%d is : %4d\n", i+1, T[i]);
    fprintf(file_ptr, "\nThe General Solution is:\n\n");
    fprintf(file_ptr, " X = ( ");
    FOR(j = 0; j < m; j++) {
        temp_sum = 0;
        FOR (i = 0; i < exist; i++)
            temp_sum += T[i] * U[i*m + j];
        fprintf(file_ptr, " %d ", temp_sum);
        FOR(i = exist; i < m; i++){
            IF(U[i*m + j])
                fprintf(file_ptr, "+ %d * t%d",
                        U[i*m + j], i+1);
        }
        IF(j < m-1)
            fprintf(file_ptr, ", ");
    }
}
```

```
    fprintf(file_ptr, "    )\n");

}

fprintf(file_ptr,
        "\n THE MATRIX U After Diagonlization is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR (j = 0; j < m; j++)
        fprintf(file_ptr, " %4d ", U[i*m + j]);
    fprintf(file_ptr, "\n");
}

fprintf(file_ptr,
        "\n THE MATRIX D After Diagonalization is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR(j = 0; j < n; j++) {
        fprintf(file_ptr, " %4d ", D[i*n + j]);

    }
    fprintf(file_ptr, "\n");
}
fprintf(file_ptr,
        "\n THE MATRIX V After Diagonlization is :\n\n");
FOR(i = 0; i < n; i++) {
    FOR (j = 0; j < n; j++)
        fprintf(file_ptr, " %4d ", V[i*n + j]);
    fprintf(file_ptr, "\n");
}

fclose(file_ptr);
}
```

Diophantine Equation Solver (2)

```
/*
 *** File "dio_2.c"
 ***
 */
#include "style.h"

*****
***** Diophantine Equation Routine 2; Theorem 3.7.
*****
***** Description: Using Algorithm 2.3, it decides if a
***** given system of linear diophantine equations
*****
*****  $XA = C$ 
*****
***** has a solution, and finds the general
***** solution when solutions exist.
****

***** Input: An  $m \times n$  integer matrix  $A$  and an integer
*****  $n$ -vector  $C$ .
*****
***** Output: The statement whether the equation has a
***** solution. If solutions exist, an output
***** of the solution.
****/


EXTERN VOID diag( INT *, INT *, INT *, INT *, INT, INT);

VOID diophantine_2
{
    INT      *A,
    INT      *U,
    INT      *V,
    INT      *D,
    INT      *C,
    INT      *T,
    INT      m,
    INT      n,
    INT      *exist
}
{
    INT      i, j, k, io, ii, q, sig_s, temp;
```

```

INT      rho, a, found;

< /**
***      Call Diagonalization Algorithm
**/


diag(A, U, V, D, m, n);

< /**
***      Find rho: Number of nonzero rows from the
***              top of D.
**/


found = 0;
FOR(i=m-1; i >= 0; i --){
    FOR(j = 0; j < n; j++)
        IF(D[i*n +j]) {
            rho = i + 1;
            found++;

            BREAK;
        }
    IF(found) BREAK;
}

< /**
***      Solve for TD = CV
**/


*exist = rho;
FOR(j = 0; j < n; j++) {
    a = 0;
    FOR(i = 0; i < n; i++)
        a += C[i] * V[i *n +j];
    T[j] = a / D[j *n +j];

    IF((C[j] % D[j*n+j]) != 0) {
        *exist = -1;
        RETURN;
    }
}
}

```

Output From a Sample Run Of Diophantine Equation Solver (2)

MATRIX A	VECTOR C	MATRIX U	MATRIX D	MATRIX V
(1)				
4	8	-1 1 0	2	1
6		3 -2 0	0	
4		-1 0 1	0	

Solution to the set of equations $TD = CV$ is

$$t_1 = 4$$

The General Solution to $XA = C$ is

$$X = (-4 + 3*t_2 + -1*t_3, 4 + -2*t_2, t_3)$$

(2)											
2	1 0 4	1 0 0 0	1 0 0	0 0 1							
6	3 -2 2	-5 0 0 1	0 -1 0	1 0 -2							
4	0 3 8	-8 1 1 1	0 0 -8	0 1 -12							
-2	5 -1	-17 4 3 1	0 0 0								

Solution to the set of equations $TD = CV$ is

$$t_1 = 2$$

$$t_2 = -8$$

$$t_3 = 12$$

The General Solution to $XA = C$ is

$$X = (-54 + -17*t_4, 12 + 4*t_4, 12 + 3*t_4, 4 + t_4)$$

(3)											
3	6 1 0	3	1								
-3		1 1	0								

Solution to the set of equations $TD = CV$ is

$$t_1 = 2$$

The General Solution to $XA = C$ is

$$X = (2 + t_2, t_2)$$

(4)											
10	15 3 -2	2	1								
14		-7 5	0								

No solutions exist to the set of equations

(5)

$$\begin{array}{rrrrr} 55 & 17 & 0 & 6 & 13 \\ -89 & & 0 & 41 & 89 \\ 41 & & 1 & 2 & 3 \end{array} \quad \begin{array}{r} -1 \\ 0 \\ 0 \end{array} \quad \begin{array}{r} 1 \\ \\ \end{array}$$

Solution to the set of equations TD = CV is

$$t_1 = -17$$

The General Solution to XA = C is

$$X = (t_3, -102 + 41*t_2 + 2*t_3, -221 + 89*t_2 + 3*t_3)$$

(6)

$$\begin{array}{rrrrr} 14 & -42 & -1 & 1 & 0 \\ 21 & & 3 & -2 & 0 \\ -35 & & 1 & 1 & 1 \\ 28 & & -2 & 0 & 0 \end{array} \quad \begin{array}{r} 0 \\ 0 \\ 0 \\ 1 \end{array} \quad \begin{array}{r} 7 \\ 0 \\ 0 \\ 0 \end{array} \quad \begin{array}{r} 1 \\ \\ \\ \end{array}$$

Solution to the set of equations TD = CV is

$$t_1 = -6$$

The General Solution to XA = C is

$$X = (6 + 3*t_2 + t_3 + -2*t_4, -6 + -2*t_2 + t_3, t_3, t_4)$$

(7)

$$\begin{array}{rrrrr} 3 & 10 & 6 & 1 & 0 \\ -3 & 14 & 15 & -23 & 1 \end{array} \quad \begin{array}{r} 1 \\ 0 \\ 0 \\ -72 \end{array} \quad \begin{array}{r} -3 \\ 1 \\ -3 \end{array} \quad \begin{array}{r} 10 \\ -3 \\ -3 \end{array}$$

No solutions exist to the set of equations

(8)

$$\begin{array}{rrrrr} 3 & 55 & 15 & 1 & 0 \\ 14 & -89 & 17 & -271 & 58 \\ 0 & 41 & & -574 & 123 \end{array} \quad \begin{array}{r} 0 \\ 489 \\ 1037 \end{array} \quad \begin{array}{r} 0 \\ -1 \\ 0 \end{array} \quad \begin{array}{r} -18 \\ 55 \\ 1 \end{array} \quad \begin{array}{r} 0 \\ -3 \\ 0 \end{array}$$

Solution to the set of equations TD = CV is

$$t_1 = -253$$

$$t_2 = -774$$

The General Solution to XA = C is

$$X = (209501 + -574*t_3, -44892 + 123*t_3, -378486 + 1037*t_3)$$

(9)

$$\begin{array}{ccccccccc} 10 & 55 & 15 & 4 & -3 & 0 & 1 & 0 & \\ 13 & -89 & 17 & -91 & 70 & 274 & 0 & -1 & \\ 0 & 41 & & -533 & 410 & 1605 & 0 & 0 & \end{array} \quad \begin{array}{cc} 1 & -487 \\ 0 & 1 \end{array}$$

Solution to the set of equations TD = CV is

$$t_1 = 15$$

$$t_2 = 7288$$

The General Solution to XA = C is

$$X = (-663148 + -533*t_3, 510115 + 410*t_3, 1996912 + 1605*t_3)$$

(10)

$$\begin{array}{ccccccccc} 1 & -2 & 5 & 5 & 1 & 0 & 0 & 0 & \\ 3 & -1 & 2 & 8 & 2 & 0 & 1 & 0 & \\ -2 & 1 & 0 & 8 & -3 & 3 & 2 & -2 & \\ 1 & 2 & -3 & & 13 & -16 & -12 & 11 & \end{array} \quad \begin{array}{cccccc} 1 & 0 & 0 & 1 & 1 & 5 \\ 0 & 1 & 0 & 0 & 3 & 10 \\ 0 & 0 & 1 & 0 & 1 & 3 \end{array}$$

Solution to the set of equations TD = CV is

$$t_1 = 5$$

$$t_2 = 37$$

$$t_3 = 129$$

The General Solution to XA = C is

$$X = (-308 + 13*t_4, 387 + -16*t_4, 295 + -12*t_4, -258 + 11*t_4)$$

Driver For Two-Variable Equation Solver

```
/***
***      File  "new_two.c"
***
*/
#include <stdlib.h>
#include <stdio.h>
#ifndef TWO_VAR_INCLUDED
#include "two_var.h"
#endif

/**
***      FILE  new_two.c
***      This file contains the main module and print functions
***          for the two-variable equation under constraints
***          routine.
***

***      The main module is the driver for the actual two
***          variable equation solver: two_var. It will get input
***          from the terminal and append the appropriate output
***          to a file.
***
**/

EXTERN VOID two_var ( INT, INT, INT, INT, INT,
                      Psi *, Psi *, Psi *, Psi *,
                      INT *, INT *, INT *, INT *
                    );
VOID print_Psi( Psi *, INT, FILE *);

/**
***      The main driver.
**/
main() {

    INT a, b, c, p, q;
    Psi s, s1, s0, s_1;
    INT min1, max1, min_1, max_1;
```

```
CHAR fname[FILENAME_MAX];
FILE * file_ptr;

printf("Enter file name for output:\n");
scanf("%s", fname);
file_ptr = fopen(fname, "a");

/***
*** Get the values of a, b, c, p and q
*/
printf("Enter the Number a:\n");
scanf("%d", &a);
printf("Enter the Number b:\n");
scanf("%d", &b);
printf("Enter the Number c:\n");
scanf("%d", &c);
printf("Enter the Number p:\n");
scanf("%d", &p);

printf("Enter the Number q:\n");
scanf("%d", &q);
fprintf(file_ptr,
        " Solutions for the Equation :\n\n %d * i - %d * j ",
        a, b );
fprintf(file_ptr, " = %d\n\n", c );
fprintf(file_ptr, "In the range    %d to %d \n\n",
        p, q );
/***
*** Call the two-variable solver.
*/
two_var ( a, b, c, p, q, &s, &s0, &s1, &s_1,
          &min1, &max1, &min_1, &max_1);

print_Psi( &s, 2, file_ptr);
print_Psi( &s1, 1, file_ptr);
IF(!s1.empty_set) {
    fprintf(file_ptr, "\n The value mu1 is: %d\n",min1);
    fprintf(file_ptr, " The value nul is: %d\n\n",max1);
}
print_Psi( &s0, 0, file_ptr);
print_Psi( &s_1, -1, file_ptr);
IF(!s_1.empty_set) {
    fprintf(file_ptr, "\n The value mu_1 is:%d\n",min_1);
```

```
        fprintf(file_ptr, " The value nu_1 is:%d\n\n",max_1);
    }
    fclose(file_ptr);
}

/***
***      PRINT_Psi
***      Prints the structure Psi with appropriate headings
***          to a file.
***/

VOID print_Psi(
    Psi * sptr,
    INT   mark,
    FILE * fptr
)
{
    IF(mark > 1)
        fprintf( fptr, " *** THE s STRUCTURE ** \n\n");

    IF(mark == 1)
        fprintf( fptr, " *** THE s1 STRUCTURE ** \n\n");
    IF(mark == -1)
        fprintf( fptr, " *** THE s_1 STRUCTURE ** \n\n");
    IF(mark == 0)
        fprintf( fptr, " *** THE s0 STRUCTURE ** \n\n");

    IF(sptr->empty_set) {
        fprintf (fptr, "      IS EMPTY !!! \n\n");
        RETURN;
    }
    fprintf(fptr, "The equation is: ((%d + %d t, %d + %d t)",
            sptr->x.a_part,sptr->x.b_part,
            sptr->y.a_part,sptr->y.b_part);
    IF(!(sptr->x.b_part == 0 && sptr->y.b_part == 0))
        fprintf(fptr, " : %d <= t <= %d )\n\n",
                sptr->lbnd_t,sptr->ubnd_t);
    ELSE
        fprintf(fptr, "}\n\n");
}
```

Two-Variable Equation Solver

```
/***
***  File  "two_var.c"
***
***/
#include <stdlib.h>
#include <math.h>
#include <values.h>
#ifndef TWO_VAR_INCLUDED
#include "two_var.h"
#endif

EXTERN VOID  echelon_reduc (INT *, INT *, INT *, INT, INT);
/***
***      FILE    two_var.c
***      Contains the module two_var. It is an implementation
***      of Algorithm 3.1 in the book.

***      This module decides if there is a solution to the
***      linear diophantine equation in two variables
***      ai - bj = c
***      that satisfies the constraints:
***      p <= i <= q and p <= j <= q
***      Input: Five integers: a, b, c, p and q.
***      Output: The solution sets s, s1, s_1, s0, and
***              the extreme values mul (min1), nul (max1),
***              mu_1 (min_1) and nu_1 (max_1).
**/


VOID two_var (
    INT    a,
    INT    b,
    INT    c,
    INT    p,
    INT    q,
    Psi   *s,
```

```

    Psi    *s0,
    Psi    *s1,
    Psi    *s_1,
    INT    *min1,
    INT    *max1,
    INT    *min_1,
    INT    *max_1
)
{
    INT    int_xi,g,t1,t2,t3,t4,t5,t6;
    DOUBLE xi;
    INT    plus, minus, diff,
    INT    U[2][2], S[2], A[2];
    INT    i,j, 10, 11, j0, j1;

    /**
    ***      STEP   1
    ***      Set the sets to NULL.
    **/


    s->empty_set    = TRUE;
    s1->empty_set   = TRUE;
    s0->empty_set   = TRUE;
    s_1->empty_set  = TRUE;

    /**
    ***      STEP   2
    ***      Check for the case when both a and b are zero.
    ***      Trivial case: c = 0 !!
    **/


    IF(a == 0 && b == 0) {
        IF( q >= p) {

            s->empty_set = FALSE;
            s->lbnd_t = p;
            s->ubnd_t = q;
            s->x.a_part = s->y.a_part = 1;
            s->x.b_part = s->y.b_part = 0;
            *s0 = *s1 = *s_1 = *s;
            *min1 = 1;
            *max1 = q-p;

            *min_1 = 1;
        }
    }
}

```

```
        *max_1 = q-p;
    }
    RETURN;
}

/***
***      STEP   3
***      Prepare the matrices and call echelon reduction.
**/


A[0] = a; A[1] = -b;

echelon_reduc(&A[0], &U[0][0], &S[0], 2, 1);

IF(S[0] < 0) {
    U[0][0] = (-U[0][0]);
    U[0][1] = (-U[0][1]);
    S[0]     = (-S[0]);
}

/***
***      STEP   4
***      Get the gcd value from matrix S, and see if it
***      divides c. If it does not, there is no solution.
**/


g = S[0];
IF(c % g != 0) {
    RETURN;
}

/***
***      STEP   5
***      Compute the values i0, i1, j0 and j1.
**/


i0 = c * U[0][0] /g;
j0 = c * U[0][1] /g;

i1 = U[1][0];
j1 = U[1][1];

/***
***      STEP   6
```

```
***      Set initial values of t1 and t2.  
**/  
  
t1 = -MAXINT;  
t2 = MAXINT;  
  
/**  
***      Check for values of i1  
***      and set t1, t2 appropriately.  
**/  
  
IF (i1 == 0) {  
    IF ( p > q ||  
        i0 > q ||  
        i0 < p ) {  
  
        RETURN;  
    }  
}  
  
IF (i1 > 0) {  
    t1 = MAX (t1, (INT) ceil((DOUBLE)(p-i0)/  
                           (DOUBLE)i1));  
    t2 = MIN (t2, (INT) floor((DOUBLE)(q-i0)/  
                           (DOUBLE)i1));  
}  
IF (i1 < 0) {  
  
    t1 = MAX (t1, (INT) ceil((DOUBLE)(q-i0)/  
                           (DOUBLE)i1));  
    t2 = MIN (t2, (INT) floor((DOUBLE)(p-i0)/  
                           (DOUBLE)i1));  
}  
  
/**  
***      Check for Values of j1  
***      And set t1, t2 appropriately.  
**/  
IF (j1 == 0) {  
    IF ( p > q ||  
        j0 > q ||  
        j0 < p ) {  
  
        RETURN;  
    }  
}
```

```
}

IF (j1 > 0 ) {
    t1 = MAX (t1, (INT) ceil((DOUBLE)(p-j0)/
                           (DOUBLE)j1));
    t2 = MIN (t2, (INT) floor((DOUBLE)(q-j0)/
                           (DOUBLE)j1));
}

IF (j1 < 0 ) {

    t1 = MAX (t1, (INT) ceil((DOUBLE)(q-j0)/
                           (DOUBLE)j1));
    t2 = MIN (t2, (INT) floor((DOUBLE)(p-j0)/
                           (DOUBLE)j1));
}

/***
***      STEP    7
***      If t1 > t2; terminate the algorithm;
***      Else, initialize the data structure s.

*/
IF( t1 > t2) {
    RETURN;
}

s->empty_set = FALSE;
s->x.a_part = i0;
s->x.b_part = i1;
s->y.a_part = j0;
s->y.b_part = j1;
s->lbind_t = t1;
s->ubnd_t = t2;

/***
***      STEP    8
***      Fill the subsets s1, s_1, s0 if i1 == j1 and
***              terminate.
*/
IF(i1 == j1) {
    IF(i0 < j0) {
        *s1 = *s;
    }
    IF(i0 > j0) {
        *s_1 = *s;
    }
}
```

```

        }

        IF(i0 == j0) {
            *s0 = *s;
        }

        IF(! (s1->empty_set) ) {
            *min1 = j0 - i0;
            *max1 = j0 - i0;
        }

        IF(! s_1->empty_set) {
            *min_1 = i0 - j0;
            *max_1 = i0 - j0;
        }

    RETURN;
}

/***
***      STEP   9
***      If the ratio of (i0-j0) and (j1-i1) is an
***          integer, Compute s0 set based on this
***          information.
***/

IF( ((i0 -j0) % (j1 -i1) ) == 0 ) {
    int_xi = (i0 -j0) / (j1 -i1);
    IF( t1 <= int_xi && int_xi <= t2) {
        s0->empty_set = FALSE;
        s0->x.a_part = i0 + i1 * int_xi;
        s0->x.b_part = 0;
        s0->y.a_part = j0 + j1 * int_xi;
        s0->y.b_part = 0;
    }
}
x1 = (DOUBLE) (i0 -j0) / (DOUBLE) (j1 - i1);

/***
***      STEP   10
***      Compute t3 and t4 based on xi value.
***/

t3 = (INT) ceil(xi - 1.0e+0);
t4 = (INT) floor(xi + 1.0e+0);

/***
***      STEP   11
*/

```

```
***      Compute t5 and t6
**/


t5 = MIN(t2,t3);
t6 = MAX(t1,t4);

/***
***      STEP    12
***      Since i1 != j1, compute the sets s1 and
***          s_1, based on the relative values.
**/


IF( i1 > j1 ) {
    IF(t5 >= t1) {
        s1->empty_set = FALSE;
        s1->x.a_part  = i0;
        s1->x.b_part  = i1;

        s1->y.a_part  = j0;

        s1->y.b_part  = j1;

        s1->lbdn_t     = t1;
        s1->ubnd_t     = t5;
    }
    IF(t2 >= t6) {
        s_1->empty_set = FALSE;
        s_1->x.a_part  = i0;
        s_1->x.b_part  = i1;

        s_1->y.a_part  = j0;
        s_1->y.b_part  = j1;

        s_1->lbdn_t     = t6;
        s_1->ubnd_t     = t2;
    }
}
IF( i1 < j1 ) {
    IF(t2 >= t6) {
        s1->empty_set = FALSE;
        s1->x.a_part  = i0;
        s1->x.b_part  = i1;

        s1->y.a_part  = j0;
        s1->y.b_part  = j1;
```

```

    s1->lbnd_t      = t6;
    s1->ubnd_t      = t2;
}
IF(t5 >= t1) {
    s_1->empty_set = FALSE;
    s_1->x.a_part  = i0;
    s_1->x.b_part  = i1;

    s_1->y.a_part  = j0;
    s_1->y.b_part  = j1;

    s_1->lbnd_t      = t1;
    s_1->ubnd_t      = t5;
}
/***
***   STEP   13
***   Set the extreme values nul, mu1,
***       nu_1 and mu_1!!
*/
IF(!s1->empty_set) {
    i0 = s1->x.a_part;
    i1 = s1->x.b_part;
    j0 = s1->y.a_part;
    j1 = s1->y.b_part;

    diff = j1 - i1;
    plus = PLUS_PART(diff);

    minus = MINUS_PART(diff);

    *min1 = (j0 - i0) + plus * s1->lbnd_t
           - minus * s1->ubnd_t;
    *max1 = (j0 - i0) + plus * s1->ubnd_t
           - minus * s1->lbnd_t;
}

IF(!s_1->empty_set) {
    i0 = s_1->x.a_part;
    i1 = s_1->x.b_part;
    j0 = s_1->y.a_part;
    j1 = s_1->y.b_part;
}

```

```
diff = i1 - j1;
plus = PLUS_PART(diff);
minus = MINUS_PART(diff);

*min_1 = (i0 - j0) + plus * s_1->lbnd_t
        - minus * s_1->ubnd_t;
*max_1 = (i0 - j0) + plus * s_1->ubnd_t
        - minus * s_1->lbnd_t;
}

/***
***      STEP    14
***      Done! Return.
*/
RETURN;
}
```

Output From A Sample Run Of the Two-Variable Solver

(1) Solutions for equation: $6*i - 4*j = 10$ range 0 to 35

THE s STRUCTURE for the equation :

$\{(5 + 2 t, 5 + 3 t) : -1 \leq t \leq 10\}$

THE s1 STRUCTURE for the equation :

$\{(5 + 2 t, 5 + 3 t) : 1 \leq t \leq 10\}$

The value mu1 is : 1 & The value nu1 is : 10

THE s0 STRUCTURE for the equation is:

$\{(5 + 0 t, 5 + 0 t)\}$

THE s_1 STRUCTURE for the equation is:

$\{(5 + 2 t, 5 + 3 t) : -1 \leq t \leq -1\}$

The value mu_1 is : 1 & The value nu_1 is : 1

(2) Solutions for equation: $3*i - 3*j = 6$ range 1 to 100

THE s STRUCTURE for the equation :

$\{(0 + 1 t, -2 + 1 t) : 3 \leq t \leq 100\}$

THE s1 STRUCTURE IS EMPTY !!!

THE s0 STRUCTURE IS EMPTY !!!

THE s_1 STRUCTURE for the equation :

$\{(0 + 1 t, -2 + 1 t) : 3 \leq t \leq 100\}$

The value mu_1 is : 2 & The value nu_1 is : 2

(3) Solutions for equation: $3*i - 7*j = 6$ range 1 to 100

THE s STRUCTURE for the equation :

$\{(-12 + 7 t, -6 + 3 t) : 3 \leq t \leq 16\}$

THE s1 STRUCTURE IS EMPTY !!!

THE s0 STRUCTURE IS EMPTY !!!

THE s_1 STRUCTURE for the equation :

$\{(-12 + 7 t, -6 + 3 t) : 3 \leq t \leq 16\}$

The value mu_1 is : 6 & The value nu_1 is : 58

(4) Solutions for equation: $-3*i - 19*j = 2$ range 1 to 100

THE s STRUCTURE for the equation is:

$$\{(12 + 19 t, 2 + 3 t) : 0 \leq t \leq 4\}$$

THE s1 STRUCTURE IS EMPTY !!!

THE s0 STRUCTURE IS EMPTY !!!

THE s_1 STRUCTURE for the equation :

$$\{(12 + 19 t, 2 + 3 t) : 0 \leq t \leq 4\}$$

The value mu_1 is : 10 & The value nu_1 is : 74

(5) Solutions for equation: $10*i - 14*j = 10$ range 1 to 100

THE s STRUCTURE for the equation:

$$\{(15 + 7 t, 10 + 5 t) : -1 \leq t \leq 12\}$$

THE s1 STRUCTURE IS EMPTY !!!

THE s0 STRUCTURE IS EMPTY !!!

THE s_1 STRUCTURE for the equation :

$$\{(15 + 7 t, 10 + 5 t) : -1 \leq t \leq 12\}$$

The value mu_1 is : 3 & The value nu_1 is : 29

(6) Solutions for equation: $10*i - 14*j = 4$ range 1 to 100

THE s STRUCTURE for the equation :

$$\{(6 + 7 t, 4 + 5 t) : 0 \leq t \leq 13\}$$

THE s1 STRUCTURE IS EMPTY !!!

THE s0 STRUCTURE IS EMPTY !!!

THE s_1 STRUCTURE for the equation:

$$\{(6 + 7 t, 4 + 5 t) : 0 \leq t \leq 13\}$$

The value mu_1 is : 2 & The value nu_1 is : 28

(7) Solutions for equation: $3*i - -3*j = 6$ range 1 to 100

THE s STRUCTURE for the equation:

$$\{(0 + 1 t, 2 + -1 t) : 1 \leq t \leq 1\}$$

THE s1 STRUCTURE IS EMPTY !!!

THE s0 STRUCTURE for the equation : $\{(1 + 0 t, 1 + 0 t)\}$

THE s_1 STRUCTURE IS EMPTY !!!

(8) Solutions for equation: $5*i - 2*j = -3$ range 1 to 100

THE s STRUCTURE for the equation:

$$\{(-3 + 2 t, -6 + 5 t) : 2 \leq t \leq 21\}$$

THE s1 STRUCTURE for the equation:

$$\{(-3 + 2 t, -6 + 5 t) : 2 \leq t \leq 21\}$$

The value mul is : 3 & The value nul is : 60

THE s0 STRUCTURE IS EMPTY !!!

THE s_1 STRUCTURE IS EMPTY !!!

(9) Solutions for equation: $55*i - 89*j = 12$ range 1 to 100

THE s STRUCTURE for the equation :

$$\{(408 + 89 t, 252 + 55 t) : -4 \leq t \leq -4\}$$

THE s1 STRUCTURE IS EMPTY !!!

THE s0 STRUCTURE IS EMPTY !!!

THE s_1 STRUCTURE for the equation:

$$\{(408 + 89 t, 252 + 55 t) : -4 \leq t \leq -4\}$$

The value mu_1 is : 20 & The value nu_1 is : 20

(10)Solutions for equation: $2*i - -3*j = 46$ range 1 to 100

THE s STRUCTURE for the equations:

$$\{(-46 + 3 t, 46 + -2 t) : 16 \leq t \leq 22\}$$

THE s1 STRUCTURE for the equation:

{(-46 + 3 t, 46 + -2 t) : 16 <= t <= 18 }

The value mul is : 2 & The value nul is : 12

THE s0 STRUCTURE IS EMPTY !!!

THE s_1 STRUCTURE for the equation:

{(-46 + 3 t, 46 + -2 t) : 19 <= t <= 22 }

The value mu_1 is : 3 & The value nu_1 is : 18

(11) Solutions for equation: -3*i - 2*j = 6 range 1 to 100

THE s STRUCTURE IS EMPTY !!!

Rational Number Primitives

```

 $\ast\ast\ast$ 
 $\ast\ast\ast \quad \text{File} \quad \text{"rational.c"}$ 
 $\ast\ast\ast$ 
 $\ast\ast\ast$ 
 $\#include \text{ "style.h"}$ 
 $\#include \text{ "two_var.h"}$ 

RATIONAL rat_div(
    RATIONAL a,
    RATIONAL b
)
{
    INT      r_n, r_d, g;
    RATIONAL r;

    r = (RATIONAL) malloc( 1, sizeof(RATIONAL_NODE) );

    r_n = a->num * b->denom;
    r_d = b->num * a->denom;

    IF( (g = gcd(r_n, r_d)) != 0 ) {
        r_n /= g; r_d /= g;
    }

    IF(SIG(r_n) == SIG(r_d)) {
        r_n = ABS(r_n);
        r_d = ABS(r_d);
    }
    ELSE {
        r_n = - (ABS(r_n));
        r_d = (ABS(r_d));
    }

    r->num = r_n;
    r->denom = r_d;

    RETURN r;
}

```

```
RATIONAL rat_mul(
    RATIONAL a,
    RATIONAL b
)
{
    INT      r_n, r_d, g;
    RATIONAL r;

    r = (RATIONAL) malloc( 1, sizeof(RATIONAL_NODE) );

    r_n = a->num * b->num;
    r_d = b->denom * a->denom;

    IF( (g = gcd(r_n, r_d)) != 0) {
        r_n /= g; r_d /= g;
    }
    IF(SIG(r_n) == SIG(r_d)) {

        r_n = ABS(r_n);
        r_d = ABS(r_d);
    }
    ELSE {
        r_n = - (ABS(r_n));
        r_d = (ABS(r_d));
    }

    r->num = r_n;
    r->denom = r_d;

    RETURN r;
}

RATIONAL rat_add(
    RATIONAL a,
    RATIONAL b
)
{
    INT      r_n, r_d, g;
    RATIONAL r;

    r = (RATIONAL) malloc( 1, sizeof(RATIONAL_NODE) );
```

```

r_d = a->denom * b->denom;
r_n = (a->num * b->denom) + (b->num * a->denom);

IF( (g = gcd(r_n, r_d)) != 0) {
    r_n /= g; r_d /= g;
}
IF(SIG(r_n) == SIG(r_d)) {
    r_n = ABS(r_n);
    r_d = ABS(r_d);
}
ELSE {
    r_n = - (ABS(r_n));
    r_d = (ABS(r_d));
}

r->num = r_n;
r->denom = r_d;

RETURN r;
}

RATIONAL rat_sub(
    RATIONAL a,
    RATIONAL b
)
{
    INT      r_n, r_d, g;

    RATIONAL r;

    r = (RATIONAL) malloc( 1, sizeof(RATIONAL_NODE) );

    r_d = a->denom * b->denom;
    r_n = (a->num * b->denom) - (b->num * a->denom);

    IF( (g = gcd(r_n, r_d)) != 0) {
        r_n /= g; r_d /= g;
    }
    IF(SIG(r_n) == SIG(r_d)) {
        r_n = ABS(r_n);
        r_d = ABS(r_d);
    }
}

```

```
    ELSE {
        r_n = - (ABS(r_n));
        r_d = (ABS(r_d));
    }

    r->num = r_n;
    r->denom = r_d;

    RETURN r;
}

RATIONAL rat_max(
    RATIONAL a,
    RATIONAL b
)
{
    RATIONAL result;

    result = rat_sub(a,b);

    IF(result->num > 0)
        *result = *a;
    ELSE
        *result = *b;

    RETURN result;
}

RATIONAL rat_min(
    RATIONAL a,
    RATIONAL b
)
{
    RATIONAL result;

    result = rat_sub(a,b);

    IF(result->num > 0)
        *result = *b;
    ELSE
        *result = *a;
```

```
    RETURN result;
}

BOOL  rat_eqt(
    RATIONAL a,
    RATIONAL b
)
{
    RATIONAL result;
    BOOL      ret = FALSE;

    result = rat_sub(a,b);
    IF(! (result->num) )  ret= TRUE;
    free(result);
    RETURN  ret;
}

BOOL  rat_grt(
    RATIONAL a,

    RATIONAL b
)
{
    RATIONAL result;
    BOOL      ret = FALSE;

    result = rat_sub(a,b);
    IF(result->num > 0)  ret= TRUE;
    free(result);
    RETURN  ret;
}

BOOL  rat_lst(
    RATIONAL a,
    RATIONAL b
)
{
    RATIONAL result;
    BOOL      ret = FALSE;

    result = rat_sub(a,b);
    IF(result->num < 0)  ret= TRUE;
    free(result);
    RETURN  ret;
}
```

```
INT gcd(
    INT x,
    INT y
)
{
    INT         a = x;
    INT         b = y;

    IF (a < 0) a = -a;
    IF (b == 0) RETURN a;

    IF (b < 0) b = -b;
    IF ((a = a % b) == 0) RETURN b;
    IF (a < 0) a = -a;
    WHILE(1) {
        IF ((b = b % a) == 0) RETURN a;
        IF ((a = a % b) == 0) RETURN b;
    }
}

INT lcm(
    INT x,
    INT y
)
{
    RETURN x == 0 || y == 0 ? 0 : ABS((x / gcd(x,y)) * y);
}
```

Driver For Fourier Elimination

```

/*
***      File  "new_fou.c"
*/
#include <stdlib.h>
#include <stdio.h>
#include "style.h"
#include "two_var.h"
****/
***** new_fou.c: This is the driver for fourier elimination
***** routine.
*****
***** This will read the input matrix A and vector C,
***** allocate storage for A, C, B and b and call
***** fourier_elim routine.
****/
EXTERN VOID  fourier_elim( RATIONAL A, RATIONAL C, INT m,
                           INT n, RATIONAL B, RATIONAL b, INT * exist,
                           FILE * file_ptr);
main() {
    INT i,j,k,m,n;
    INT exist;
    RATIONAL A, B, C, b;
    CHAR fname[FILENAME_MAX];
    FILE * file_ptr;

    printf("Enter file name for output:\n");
    scanf("%s", fname);
    file_ptr = fopen(fname, "a");
    printf("Enter the Number m:\n");
    scanf("%d", &m);
    printf("The Number m is : %d\n",m);
    printf("Enter the Number n:\n");
    scanf("%d", &n);
    printf("The Number n is: %d\n",n);
    A = (RATIONAL)malloc(m*n,sizeof(RATIONAL_NODE));
    C = (RATIONAL)malloc(n,sizeof(RATIONAL_NODE));
    B = (RATIONAL)malloc(m,sizeof(RATIONAL_NODE));
    b = (RATIONAL)malloc(m,sizeof(RATIONAL_NODE));

    FOR(i = 0; i < m; i++) {

```

```

        printf("Enter Row %d of matrix A:\n",i);
FOR(j = 0; j < n; j++) {
    scanf("%d",&((A+i*n+j)->num));
    (A+i*n+j)->denom = 1;
}
printf("Enter vector C:\n",i);
FOR(i = 0; i < n; i++) {
    scanf("%d",&((C+i)->num));
    (C+i)->denom = 1;
}
fprintf(file_ptr,
        "\n THE MATRIX A Before Fourier Elimination is :\n\n");
FOR(i = 0; i < m; i++) {
    FOR (j = 0; j < n; j++)
        fprintf(file_ptr, " %d ", (A[ i * n + j]).num);
    fprintf(file_ptr, "\n");
}

fprintf(file_ptr,
        "\n THE VECTOR C Before Fourier Elimination is :\n\n");
FOR(i = 0; i < n; i++)
    fprintf(file_ptr, " %d ", (C[i]).num);
fprintf(file_ptr, "\n");

exist = -1;
fourier_elim(A, C, m, n, B, b, &exist, file_ptr);
IF(exist < 0)
    fprintf(file_ptr, "\nThere are no solutions to\
                           this set of inequalities\n\n");
fclose(file_ptr);
}

```

Fourier Elimination

```

#include <stdlib.h>
#include <stdio.h>
#include "style.h"
#include "two_var.h"

VOID solve (RATIONAL t, RATIONAL q, INT r, INT s ,
            RATIONAL B, RATIONAL b, INT * exist,
            FILE *file_ptr);

*****
*****
*****
***** fourier.c : This routine implements Fourier
***** Elimination Algorithm 3.2 from the book.
*****
***** Description: It decides if a system of linear
***** inequalities of form

*****
          XA <= C
*****
          has a real solution X. If there is one,
          it also finds the solution set.
*****
          Input: An m X n rational matrix A and a rational
          n-vector C.
*****
          Output: The statement whether or not the system has a
          real solution, and functions b(i) and B(i) in
          case it does.
*****
          Steps: Follows the book very closely.
*****
*****
***** The major entry point: fourier_elim:
*****
***** This routine creates temporary work matrices t and q
***** and calls the real solver, solve.
*****
*/

```

```

VOID fourier_elim(
    RATIONAL      A,

```

```

RATIONAL      C,
INT           m,
INT           n,
RATIONAL      B,
RATIONAL      b,
INT           *exist,
FILE          *file_ptr
)
{
    INT       r, s;
    RATIONAL   t, q;
    INT        i, j;
    INT        n1, n2;

    /**
     *** STEP 1
     */
    r = m; s = n;

    t = (RATIONAL) calloc( r *s , sizeof(RATIONAL_NODE) );
    q = (RATIONAL) calloc( s      , sizeof(RATIONAL_NODE) );
    FOR (j = 0; j < n; j ++){
        FOR(i = 0; i < m; i++)
            t[i*n+j] = A[i*n+j];
        q[j] = C[j];
    }

    solve (t, q, r, s, B, b, exist, file_ptr);

    free( (CHAR *) t);
    free( (CHAR *) q);
}

/**
**** Solve: This is a recursive routine: at each
**** invocation, it works with matrices t and q to
**** produce the solutions b and B.
**** The matrices t and q are changed in each call,
**** and the number of rows and columns of t and
**** the number of elements in q are adjusted in

```

```

***** each call.
*****
****

VOID solve(
    RATIONAL t,
    RATIONAL q,
    INT r,
    INT s,
    RATIONAL B,
    RATIONAL b,
    INT *exist,
    FILE *file_ptr
)
{
    INT n1, n2, j, j1, j2, k, i, s_p, r_p;
    RATIONAL_NODE temp,b1,B1;
    RATIONAL temp_ptr;
    RATIONAL t_prime,q_prime;

    INT last = s-1;

    /**
     *** STEP 2
     ***
     ***      Rearrange and find n1 and n2
     ***
     ***      Find n2
     **/


    FOR(j1 = 0, j2 = last; j1 < j2; ) {
        WHILE( j1 < j2 && (t[(r-1)*s+j1].num) ) ++j1;
        WHILE( j1 < j2 && !(t[(r-1)*s+j2].num) ) --j2;
        IF (j1 < j2) {
            FOR(i = 0; i < r; i++) {
                temp = t[i*s+j1];
                t[i*s+j1] = t[i*s+j2];
                t[i*s+j2] = temp;
            }

            temp = q[j1];
            q[j1] = q[j2];
            q[j2] = temp;
        }
    }
}

```

```

FOR( k = 0; k <= last && (t[(r-1)*s+k].num) , k++);
n2 = k-1;

/***
*** Find n1
***/

FOR(j1 = 0, j2 = n2; j1 < j2; ) {
    WHILE( j1 < j2 && (t[(r-1)*s+j1].num) > 0) ++j1;
    WHILE( j1 < j2 && (t[(r-1)*s+j2].num) < 0) --j2;
    IF (j1 < j2) {
        FOR(i = 0; i < r; i++) {
            temp = t[i*s+j1];
            t[i*s+j1] = t[i*s+j2];
            t[i*s+j2] = temp;
        }
        temp = q[j1];
        q[j1] = q[j2];
        q[j2] = temp;
    }
}
FOR( k = 0; k <= n2 && (t[(r-1)*s+k].num)>0 ; k++);
n1 = k-1;

/***
*** STEP 3
***/

FOR(j1 = 0; j1 <= n2; j1++) {
    temp_ptr = rat_div(q+j1,t+(r-1)*s+j1);
    q[j1] = *temp_ptr;
    free(temp_ptr);
    FOR(i = 0; i < r-1; i++){
        temp_ptr = rat_div(t+i*s+j1,t+(r-1)*s+j1);
        t[i*s+j1] = *temp_ptr;
        free(temp_ptr);
    }
}

/***
*** STEP 4
**/


IF( n2 <= n1)

```

```

    printf("b is -INFINITY\n");
    IF( n1 < 0)
        printf("B is INFINITY\n");

    /**
     *** STEP 5
     *** If r = 1, solve for the last unknown and
     *** return.
     **/


IF ( r <= 1) {
    B1 = q[0];
    IF(n1 > 0) {
        FOR(i = 0; i <= n1; i++) {
            temp_ptr = rat_min(&B1, q+i);
            B1 = *temp_ptr;
            free(temp_ptr);
        }
    }

    b1 = q[n2];
    IF(n2 > n1+1) {
        FOR(i = n2; i >= n1+1; i--) {
            temp_ptr = rat_max(&b1, q+i);
            b1 = *temp_ptr;
            free(temp_ptr);
        }
    }
    IF(rat_grt(&b1,&B1) ){
        fprintf(file_ptr,
                " No Solutions, b1 > B1 \n");
        fprintf(file_ptr,
                " b1 is %d/%d and B1 is %d/%d\n",
                b1.num,b1.denom,B1.num,B1.denom);
        RETURN;
    }
    FOR(i = n2+1; i < s; i++){
        IF(q[i].num < 0) {
            fprintf(file_ptr,
                    " No Solutions, q[%d] < 0\n", i);
            RETURN;
        }
    }
    *exist = 1;
    fprintf(file_ptr, " %d/%d <= X1 <= %d/%d\n",

```

```

        b1.num,b1.denom,B1.num,B1.denom);
*(b+r-1) = b1;

*(B+r-1) = B1;
RETURN;
}

/***
*** STEP 6
***/

s_p = s - (n2+1) + (n1+1)*(n2-n1);
IF(s_p <= 0) {
*exist = 0;
fprintf(file_ptr,
" All real vectors between bi and Bi \
m >= i >= r are Solutions\n");
RETURN;
}

/***
*** STEP 7
*** Print the inequality in terms of the
*** unknowns.
***/
r_p = r-1;
fprintf(file_ptr, "\n");
fprintf(file_ptr, " MAX( ");
FOR(i = n1+1; i <= n2; i++){
FOR(j = 0; j < r_p ;j++){
IF(t[j*s + i].num > 0 )
fprintf(file_ptr, " -%d/%d X%d ",
(t[j*s + i]).num,
(t[j*s + i]).denom,j+1);
ELSE IF(t[j*s + i].num < 0 ){
IF(j > 0) fprintf(file_ptr, " + ");
fprintf(file_ptr, " %d/%d X%d ",
ABS((t[j*s + i]).num),
(t[j*s + i]).denom,j+1);
}
}
IF(q[i].num > 0) fprintf(file_ptr, "+");
IF(q[i].num != 0)
fprintf(file_ptr, " %d/%d ", q[i].num,q[i].denom);
IF(i != s_p -1) fprintf(file_ptr, " , ");
}
}

```

```

}

fprintf(file_ptr, ") <= X%d <= MIN( " ,r);

FOR(i = 0; i <= n1; i++) {
    FOR(j = 0; j < r_p ;j++) {
        IF(t[j*s + i].num > 0 )
            fprintf(file_ptr, " -%d/%d X%d ",
                    (t[j*s + i]).num,
                    (t[j*s + i]).denom,j+1);
        ELSE IF(t[j*s + i].num < 0 ){
            IF(j > 0) fprintf(file_ptr, " + ");
            fprintf(file_ptr, " %d/%d X%d ",
                    ABS((t[j*s + i]).num),
                    (t[j*s + i]).denom,j+1);
        }
    }
    IF(q[i].num > 0) fprintf(file_ptr, "+");
    IF(q[i].num != 0)
        fprintf(file_ptr, " %d/%d ", q[i].num,q[i].denom);

        IF(i != n1) fprintf(file_ptr, " , ");
    }
    fprintf(file_ptr, ")\\n");

/***
***   STEP7 (Continued)
***   evaluate the new values of t and q.
**/


t_prime=(RATIONAL)malloc(r_p*s_p,sizeof(RATIONAL_NODE));
q_prime=(RATIONAL)malloc(s_p,sizeof(RATIONAL_NODE));
FOR(i = 0; i < r_p; i++) {
    j = 0;
    FOR(j1 = 0; j1 <= n1; j1++) {
        FOR(j2 = n1+1; j2 <= n2; j2++) {
            temp_ptr = rat_sub(t+i*s+j1, t+i*s+j2);
            t_prime[ i*s_p + j] = *temp_ptr;
            j++;
            free(temp_ptr);
        }
    }
    FOR(j1 = n2+1; j1 < s; j1++)
        t_prime[ i*s_p + j++ ] = t[ i*s+j1];
}

```

```
j = 0;
FOR(j1 = 0; j1 <= n1; j1++) {
    FOR(j2 = n1+1; j2 <= n2; j2++) {
        temp_ptr = rat_sub(q+j1, q+j2);
        q_prime[j] = *temp_ptr;
        j++;
        free(temp_ptr);
    }
}
FOR(j2 = n2+1; j2 < s; j++, j2++)
    q_prime[j] = q[j2];
/***
*** STEP7 (Continued)
***
*** If s_p (newly calculated s) is not more than
*** original s, the old s and t can be used.
*** Reuse them.
***
*** If not, reallocate storage for t and s!!
**

IF(s_p > s ) {
    free((CHAR *) t);
    free((CHAR *) q);
    t = (RATIONAL)malloc(r_p*s_p,sizeof(RATIONAL_NODE));
    q = (RATIONAL)malloc(s_p,sizeof(RATIONAL_NODE));
}
FOR(i = 0; i < r_p; i++)
    FOR(j = 0; j < s_p; j++)
        t[i*s_p+j] = t_prime[i*s_p +j];
FOR(j = 0; j < s_p; j++)
    q[j] = q_prime[j];
free( (CHAR *) t_prime);
free( (CHAR *) q_prime);
solve(t, q, r_p, s_p, B, b, exist,file_ptr);
}
```

Output From A Sample Run Of Fourier Elimination Solver

	ORIG MATRIX A	ORIG VECTOR C
(1)	$\begin{array}{rrrr} 2 & -3 & 1 & -2 \\ -11 & 2 & 3 & 0 \end{array}$	$\begin{array}{rrrr} 3 & -5 & 4 & -3 \\ -53/5 & & & \end{array}$
	$\text{MAX}(2/11 X_1 -3/11) \leq X_2 \leq \text{MIN}(1/3 X_1 +4/3, -3/2 X_1 -5/2)$ $-53/5 \leq X_1 \leq -3/2$	
(2)	$\begin{array}{rrr} -1 & 2 & 0 \\ 1 & 0 & -10 \end{array}$	$\begin{array}{rrr} 0 & 5 & -23 \\ 23/10 & & \end{array}$
	$\text{MAX}(23/10) \leq X_2 \leq \text{MIN}(1/1 X_1)$ $23/10 \leq X_1 \leq 5/2$	
(3)	$\begin{array}{rrrr} -1 & 1 & 1 & 1 \\ 0 & 0 & -600 & 600 \end{array}$	$\begin{array}{rrrrr} 0 & 100 & -200 & 300 \\ & & & & \end{array}$
	$\text{MAX}(1/600 X_1 +1/3) \leq X_2 \leq \text{MIN}(-1/600 X_1 +1/2)$ $0/1 \leq X_1 \leq 50/1$	
(4)	$\begin{array}{rrr} 2 & 3 & 5 \\ 3 & -1 & 0 \end{array}$	$\begin{array}{rrr} 100 & 2 & 40 \\ & & \end{array}$
	$\text{MAX}(3/1 X_1 -2/1) \leq X_2 \leq \text{MIN}(-2/3 X_1 +100/3)$ <i>b1 is -INFINITY</i> $-INFINITY \leq X_1 \leq 8/1$	
(5)	$\begin{array}{rrrrr} 2 & -1 & 2 & -1 & 0 \\ 3 & 1 & 1 & 0 & -1 \\ -1 & -1 & 1 & 0 & 0 \end{array}$	$\begin{array}{rrrrr} 0 & & & & \\ 3 & 2 & 4 & 0 & 0 \\ & & & & \end{array}$
	$\text{MAX}(2/1 X_1 +3/1 X_2 -3/1) \leq X_3 \leq \text{MIN}(-2/1 X_1 -1/1 X_2 +4/1,$ $1/1 X_1 -1/1 X_2 +2/1)$ $\text{MAX}(0/1) \leq X_2 \leq \text{MIN}(-1/1 X_1 +7/4, -2/1 X_1 +4/1,$ $-1/4 X_1 +5/4, 1/1 X_1 +2/1)$ $0/1 \leq X_1 \leq 7/4$	

Driver For Linear Dependence Algorithm

```
/*
***      File "new_gen.c"
*/
#include <stdlib.h>
#include <stdio.h>
#ifndef TWO_VAR_INCLUDED
#include "two_var.h"
#endif
/**
 ***
 ***      FILE    new_gen.c
 ***
 ***      This file contains the main module and print
 ***      functions for the general linear dependence
 ***      equation solver.
 ***
 ***      This reads the matrices A, B, P, Q,
 ***
 ***      and the vectors p0, q0, a0, b0, and calls
 ***      the actual solver.
 */
EXTERN VOID lin_dep( INT *p0, INT *P, INT *q0, INT *Q,
                     INT *A, INT *a0, INT *B, INT *b0, INT m, INT n,
                     INT *exist, FILE *file_ptr ,INT *dir);

/**
***      The main driver.
*/
main() {
    INT m,n,i, j, k;
    INT *p0, *q0, *P, *Q, *A, *B, *a0, *b0;
    Psi *s, *s1, *s0, *s_1;
    INT *min1, *max1, *min_1, *max_1;
    INT exist, *dir;

    CHAR fname[FILENAME_MAX];
    FILE * file_ptr;
```

```

printf("Enter file name for output:\n");
scanf("%s", fname);
file_ptr = fopen(fname, "a");

/**
*** Get the values of m and n.
*/
printf("Enter the Number m:\n");
scanf("%d", &m);
printf("Enter the Number n:\n");
scanf("%d", &n);
/**
*** Allocate Structures.
*/
A = (INT *) calloc( m*n, sizeof(INT));
a0 = (INT *) calloc( n, sizeof(INT));
B = (INT *) calloc( m*n, sizeof(INT));
b0 = (INT *) calloc( n, sizeof(INT));

P = (INT *) calloc( m*m, sizeof(INT));
p0 = (INT *) calloc( m, sizeof(INT));
Q = (INT *) calloc( m*m, sizeof(INT));
q0 = (INT *) calloc( m, sizeof(INT));
dir= (INT *) calloc( m, sizeof(INT));

/**
*** Get the input matrices.
*/
FOR(i = 0; i < m; i++) {
    printf("Enter Row %d of matrix A\n",i);
    FOR(j = 0; j < n; j++)
        scanf("%d",A+i*n+j);
}
printf("Enter the a0 vector:\n");
FOR(j = 0; j < n; j++)
    scanf("%d",a0+j);
FOR(i = 0; i < m; i++) {
    printf("Enter Row %d of matrix B\n",i);
    FOR(j = 0; j < n; j++)
        scanf("%d",B+i*n+j);
}
printf("Enter the b0 vector:\n");
FOR(j = 0; j < n; j++)

```

```
scanf("%d",b0+j);

FOR(i = 0; i < m; i++) {
    printf("Enter Row %d of matrix P\n",i);
    FOR(j = 0; j < m; j++)
        scanf("%d",P+i*m+j);
}

printf("Enter the p0 vector:\n");
FOR(j = 0; j < m; j++)
    scanf("%d",p0+j);
FOR(i = 0; i < m; i++) {
    printf("Enter Row %d of matrix Q\n",i);
    FOR(j = 0; j < m; j++)
        scanf("%d",Q+i*m+j);
}

printf("Enter the q0 vector:\n");
FOR(j = 0; j < m; j++)
    scanf("%d",q0+j);
printf("Enter a direction vector:\n");

FOR(j = 0; j < m; j++)
    scanf("%d",dir+j);
fprintf(file_ptr,
        "\n THE MATRIX A is :\n\n");
FOR(i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
        fprintf(file_ptr, " %4d ", A[i*n + j]);
    fprintf(file_ptr, "\n");
}
fprintf(file_ptr,
        "\n THE VECTOR a0 is :\n\n");
for (j = 0; j < n; j++)
    fprintf(file_ptr, " %4d ", a0[j]);
fprintf(file_ptr, "\n");
fprintf(file_ptr,
        "\n THE MATRIX B is :\n\n");
FOR(i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
        fprintf(file_ptr, " %4d ", B[i*n + j]);
    fprintf(file_ptr, "\n");
}
fprintf(file_ptr,
        "\n THE VECTOR b0 is :\n\n");
for (j = 0; j < n; j++)
    fprintf(file_ptr, " %4d ", b0[j]);
```

```

fprintf(file_ptr, "\n");

fprintf(file_ptr,
        "\n THE MATRIX P is :\n\n");
FOR(i = 0; i < m; i++) {
    for (j = 0; j < m; j++)
        fprintf(file_ptr, " %4d ", P[i*m + j]);
    fprintf(file_ptr, "\n");
}

fprintf(file_ptr,
        "\n THE VECTOR p0 is :\n\n");
for (j = 0; j < m; j++)
    fprintf(file_ptr, " %4d ", p0[j]);
fprintf(file_ptr, "\n");

fprintf(file_ptr,
        "\n THE MATRIX Q is :\n\n");

FOR(i = 0; i < m; i++) {
    FOR (j = 0; j < m; j++)
        fprintf(file_ptr, " %4d ", Q[i*m + j]);
    fprintf(file_ptr, "\n");
}
fprintf(file_ptr,
        "\n THE VECTOR q0 is :\n\n");
for (j = 0; j < m; j++)
    fprintf(file_ptr, " %4d ", q0[j]);
fprintf(file_ptr, "\n");
fprintf(file_ptr,
        "\n The input direction vector is :\n\n");
for (j = 0; j < m; j++)
    fprintf(file_ptr, " %4d ", dir[j]);
fprintf(file_ptr, "\n");

/***
***      Call the general linear dependence routine.
***      ***
***      */
lin_dep( p0, P, q0, Q, A, a0, B, b0, m, n, &exist,
         file_ptr, dir);

```

```
IF(exist == -1)
    fprintf(file_ptr,
    "\n There is no dependence in this set.\n");
fclose(file_ptr);
}
```

Linear Dependence Algorithm

```

/***
***   File "lin_dep.c"
**/
#include <stdio.h>
#include <values.h>
#include "two_var.h"

/**
***   FILE    lin_dep.c
***  

***   Contains the module lin_dep. This is the
***   implementation of Algorithm 5.1 in the book.
***  

***   This module decides if there is a dependence
***   between two statements due to two variables
***   in a perfect nest of m loops.
***  

***  

***   Dependence due to two elements of an array of
***   dimension n is checked.
***  

***  

***   Input : m-vectors p0 and q0 (lower & upper limits),
***           m X m matrices P and Q (L and U matrices),
***           m X n integer matrices A and B giving
***           the coefficients of the two variables:
***           X(AI+a0) and X(BI+b0),
***           n vectors a0 and b0.
***  

***  

***   Output: The dependence information for the
***           two statements S and T.
***  

***  

***   Steps : The vectors P, Q, A and B are checked
***           to see if
***           1. P = Q (regular loop nest) and
***              A = B (uniform dependence)
***              then Algorithm 5.2 is used.
***           2. P = Q = Im (rectangular loop nest) and
***              A and B are m X m diagonal matrices
***              then Algorithm 5.3 is used.
***  

**/
```

```
***  
***  
***      External Function declarations:  
***  
**/  
  
EXTERN VOID echelon_reduc( INT *A, INT *U, INT *S,  
    INT m, INT n);  
EXTERN VOID two_var( INT a, INT b, INT c, INT p,  
    INT q, Psi   *s, Psi   *s0, Psi   *s1, Psi   *s_1,  
    INT *min1, INT *max1, INT *min_1, INT *max_1 );  
EXTERN VOID diophantine_1( INT *A, INT *U, INT *S,  
    INT *C, INT *T, INT m, INT n, INT *exist);  
***  
***  
***      Local Function declarations:  
***  
**/  
  
  
VOID uniform_dep( INT *p0, INT *P, INT *q0, INT *A,  
    INT *a0, INT *b0, INT m, INT n, INT *exist,  
    FILE *file_ptr);  
VOID diagonal_dep( INT *p0, INT *q0, INT *A, INT *a0,  
    INT *B, INT *b0, INT m, INT *exist,  
    FILE *file_ptr, INT *dir);  
****  
****      FUNCTION DEFINTIONS:  
****  
**/  
  
EXTERN VOID lin_dep(  
    INT      *p0,  
    INT      *P,  
    INT      *q0,  
    INT      *Q,  
    INT      *A,  
    INT      *a0,  
    INT      *B,  
    INT      *b0,  
    INT      m,  
    INT      n,  
    INT      *exist,  
    FILE    *file_ptr,  
    INT      *dir
```

```

)
{
    INT      *AB;
    INT      *U, *U1, *U2, *T;
    INT      *U1P, *U2P, *U1Q, *U2Q;
    INT      *I, *J, *D, *DIR;
    INT      *S;
    INT      i,j,k, a, M, rho, found, l1, nr;
    INT      a1,a2,a3,a4,mint,maxt;
    BOOL     equal = TRUE;
    BOOL     equal1 = TRUE;
    BOOL     uniform = TRUE;
    BOOL     diagonal = TRUE;
    BOOL     lex_pos;
    BOOL     lex_neg;

    /** Is this a regular loop nest: is P = Q ?      ***/
    /** In addition, is this a                         ***/
    /** rectangular loop nest: is P = Q = Im ?       **/


FOR(i = 0; i < m && equal; i++)
    FOR(j = 0; j < m; j++) {
        IF(P[i*m + j] != Q[i*m + j]) {
            equal = FALSE;
            equal1 = FALSE;
            BREAK;
        }
        IF(equal1) {
            IF(i == j && P[i*m + j] != 1)
                equal1 = FALSE;
            IF(i != j && P[i*m + j] != 0)
                equal1 = FALSE;
        }
    }
    IF(equal) {
        /**The loop nest is regular. Is dependence uniform?***/

        FOR(i = 0; i < m && uniform; i++)
            FOR(j = 0; j < n; j++) {
                IF(A[i*n + j] != B[i*n + j]) {
                    uniform = FALSE;
                    BREAK;
                }
            }
    }
}

```

```
IF(!uniform && equal1) {
    /** Are A and B diagonal matrices?**/

    IF(m != n) diagonal = FALSE;
    FOR(i = 0; i < m && diagonal; i++) {
        FOR(j = 0; j < m && j != i; j++) {
            IF((A[i*m + j] != B[i*m + j]) ||
               (A[i*m + j] != 0) ) {
                diagonal = FALSE;
                BREAK;
            }
        }
    } /* IF equal1 */
    ELSE IF(!uniform) diagonal = FALSE;
}
ELSE {
    uniform = FALSE;
    diagonal = FALSE;
}

IF(uniform) {
    /** Perform Uniform Dependence Testing **/
    fprintf(file_ptr, " Uniform Dependence Testing\n");
    uniform_dep( p0, P, q0, A, a0, b0, m, n,
                  exist, file_ptr);
}

ELSE IF (diagonal) {
    /** Perform Diagonal Dependence Testing **/
    fprintf(file_ptr, " Diagonal Dependence Testing\n");
    diagonal_dep( p0, q0, A, a0, B, b0, m,
                  exist, file_ptr, dir);
}

ELSE {
    /** Perform General Dependence Testing **/
    fprintf(file_ptr, " General Dependence Testing\n");
    /**
     ***
     *** This is the general dependence equation solver.
     *** It implements Algorithm 5.1 in the book.
     ***
     *** Steps: Follows the book very closely.
     ***

```

```

**/
/** STEP 1 of Algorithm 5.1 **/


AB = (INT *) calloc(2*m*n, sizeof (INT));
FOR(j = 0; j < n; j++) {
    FOR(i = 0; i < m ; i++) {
        AB[i*n + j] = A[i*n + j];
        AB[(i+m)*n + j] = -B[i*n + j];
    }
}

U = (INT *) calloc(4*m*m, sizeof(INT));
S = (INT *) calloc(2*m*n, sizeof(INT));
T = (INT *) calloc(2*m, sizeof(INT));
U1= (INT *) calloc(2*m*m, sizeof(INT));
U2= (INT *) calloc(2*m*m, sizeof(INT));
U1P = (INT *) calloc(2*m*m, sizeof(INT));
U1Q = (INT *) calloc(2*m*m, sizeof(INT));
U2Q = (INT *) calloc(2*m*m, sizeof(INT));
U2P = (INT *) calloc(2*m*m, sizeof(INT));


I = (INT *) calloc(m, sizeof(INT));
J = (INT *) calloc(m, sizeof(INT));
D = (INT *) calloc(m, sizeof(INT));
DIR = (INT *) calloc(m, sizeof(INT));

echelon_reduc( AB, U, S, 2*m, n);

/** STEP 4 of Algorithm 5.1 **/
/***
*** Find rho: Number of nonzero rows from the top of S.
***
*** Also, find M: the number of undetermined variables.
*** If M is > 1, then the equation is difficult to
*** solve, and a dependence will be assumed.
*** */
found = 0;
FOR(i=2*m-1; i >= 0; i--) {
    FOR(j = 0; j < n; j++) {
        IF(S[i*n + j]) {
            rho = i + 1;
            found++;
            BREAK;
        }
    }
}

```

```

    IF(found)  BREAK;
}
M = 2*m - rho;
*exist = rho;
IF(M >= 2) {
    *exist = 0;
    fprintf(file_ptr,
        " Possible Dependence.\n");
    goto free_ret;
}

/** STEPS 2 & 3 of Algorithm 5.1 ***/
/***
*** Solve for TS = C
**/


l1 = -1;
FOR(j = 0; j < n; j++) {
    IF(S[j]) {

        l1 = j;
        BREAK;
    }
}
nr = 0;
FOR(j = l1; j < n; j++) {
    a = 0;
    FOR(i = 0; i < nr; i++)
        a += T[i] * S[i*n + j];
    IF(S[nr*n + j]) {
        T[nr] = (b0[j-1]-a0[j-1]-a) / S[nr*n + j];
        IF((b0[j-1]-a0[j-1]-a) % S[nr*n + j]) != 0 {
            *exist = -1;
            goto free_ret;
        }
        nr++;
    }
    ELSE{
        IF(!(b0[j-1]-a0[j-1]-a)) {
            *exist = -1;
            goto free_ret;
        }
    }
}
/** STEP 5 of Algorithm 5.1 ***/

```

```

/*
 ***
 *** Set U1 and U2 as parts of U.
 ***
 */

FOR(i = 0; i < 2*m; i++)
  FOR( j = 0; j < m ; j++) {
    U1[i*m + j] = U[i*2*m + j];
    U2[i*m + j] = U[i*2*m + m + j];
  }

/** STEP 6 of Algorithm 5.1
 ***
 *** Solve For:
 ***
 *** p0 <= t(U1*P)
 ***           t(U1*Q) <= q0
 ***

***

*** p0 <= t(U2*P)
***           t(U2*Q) <= q0
***

*/
maxt = MAXINT;
mint = -MAXINT;
FOR(j = 0; j < m; j++) {
  FOR(i = 0; i < 2*m; i++) {
    U1P[i*m+j] = 0 ;
    U2P[i*m+j] = 0 ;
    U1Q[i*m+j] = 0 ;
    U2Q[i*m+j] = 0 ;
  }
  FOR(i = 0; i < 2*m; i++) {
    FOR(k = 0; k < m; k++) {
      U1P[i*m+j] += U1[i*m+k] * P[k*m+j];
      U1Q[i*m+j] += U1[i*m+k] * Q[k*m+j];
      U2P[i*m+j] += U2[i*m+k] * P[k*m+j];
      U2Q[i*m+j] += U2[i*m+k] * Q[k*m+j];
    }
  }
}
FOR(i = 0; i < m; i++) {

```

```

a1 = 0;
a2 = 0;
a3 = 0;
a4 = 0;
FOR(j = 0; j < 2*m-M; j++) {
    a1 += U1P[j*m + i] * T[j];
    a2 += U1Q[j*m + i] * T[j];
    a3 += U2P[j*m + i] * T[j];
    a4 += U2Q[j*m + i] * T[j];
}
IF(M == 0 &&
   (p0[i] > a1 || a2 > q0[i] ||
    p0[i] > a3 || a4 > q0[i])) {
    *exist = -1;
    goto free_ret;
}
IF(M != 0) {
    mint = MAX(mint, ((p0[i]-a1)/(U1P[(2*m-1)*m+i])) );
    mint = MAX(mint, ((p0[i]-a3)/(U2P[(2*m-1)*m+i])) );

    maxt = MIN(maxt, ((q0[i]-a2)/(U1Q[(2*m-1)*m+i])) );
    maxt = MIN(maxt, ((q0[i]-a4)/(U2Q[(2*m-1)*m+i])) );
}

/** STEPS 6 & 7 of Algorithm 5.1 ***/
/*
 ***
 *** Find I = T*U1 and J = T*U2, and the dependence of
 *** T on S and of S on T.
 ***
 */
FOR(i = 0; i < m; i++) {
    FOR(j = 0; j < 2*m-M; j++) {
        I[i] += U1P[j*m + i] * T[j];
        J[i] += U2P[j*m + i] * T[j];
    }
}
IF(M == 0) {
    lex_pos = FALSE;
    lex_neg = TRUE;
    FOR(i = 0; i < m; i++) {
        IF(I[i] > J[i]) {
            lex_neg = FALSE;
            lex_pos = TRUE;
        }
    }
}

```

```

        BREAK;
    }
    IF(I[i] < J[i]) {
        lex_pos = FALSE;
        BREAK;
    }

}
ELSE {
    lex_pos = FALSE;
    lex_neg = FALSE;
    FOR(i = mint; i <= maxt; i++) {
        equal = TRUE;
        FOR(j = 0; j < m; j++) {
            a1 = (I[j] + (U1[(2*m-1)*m + j] * i));
            a2 = (J[j] + (U2[(2*m-1)*m + j] * i));
            IF( a1 > a2) {
                lex_pos = TRUE;

                equal = FALSE;
                BREAK;
            }
            IF( a1 < a2) {
                lex_neg = TRUE;
                equal = FALSE;
                BREAK;
            }
        }
        IF(equal)
            lex_neg = TRUE;
        IF(lex_pos && lex_neg) BREAK;
    }
}

IF(lex_pos){
    fprintf(file_ptr,
            " Dependence of S on T is nonempty.\n");
}
IF(lex_neg){
    fprintf(file_ptr,
            " Dependence of T on S is nonempty.\n");
}

/** STEP 8 of Algorithm 5.1 ***/
/**
```

```
***  
***  $D = T^*U_2 - T^*U_1 = J - I$   
*** Find the distance vector  $D$  and  
*** the direction vector  $DIR$ .  
***  
**/  
IF(M == 0) {  
    fprintf(file_ptr,  
        " The Components Of The D Vector are:");  
    FOR(i = 0; i < m; i++) {  
        IF(lex_pos)  
            D[i] = I[i] - J[i];  
        ELSE  
            D[i] = J[i] - I[i];  
        DIR[i] = SIG(D[i]);  
    }  
    fprintf(file_ptr, "\n");  
}  
  
ELSE {  
    FOR(i = mint; i <= maxt; i++) {  
        fprintf(file_ptr,  
            " The Components of D for t = %d are:", i);  
        FOR(j = 0; j < m; j++) {  
            a1 = (U1[(2*m - 1) * m + j]) * i + I[j];  
            a2 = (U2[(2*m - 1) * m + j]) * i + J[j];  
            IF(lex_pos)  
                D[j] = a1 - a2;  
            ELSE  
                D[j] = a2 - a1;  
            fprintf(file_ptr, " %d ", D[j]);  
            DIR[j] = SIG(D[j]);  
        }  
        fprintf(file_ptr, "\n");  
    }  
}  
  
/** STEP 9 of Algorithm 5.1 **/  
free_ret:  
    free((CHAR *) U);  
    free((CHAR *) S);  
    free((CHAR *) T);  
    free((CHAR *) I);  
    free((CHAR *) J);
```

```

        free((CHAR *) D);
        free((CHAR *) DIR);
        free((CHAR *) U1);
        free((CHAR *) U2);
        free((CHAR *) U1P);
        free((CHAR *) U1Q);
        free((CHAR *) U2P);
        free((CHAR *) U2Q);
        RETURN;
    }
}

/****
*****
***** This is the routine to solve the dependence equation
***** in a regular loop nest ( $P = Q$ ) with
*****  $A \approx B$ .
*****
***** Input: The m vectors  $p_0$  and  $q_0$ ,
****

***** m X m matrix P
***** m X n matrix A,
***** n vectors  $a_0$  and  $b_0$ .
***** the integers m and n and a pointer to
***** the result flag.
****

***** Output: The solution to the dependence equation.
*****
***** Steps: Follows the book very closely.
*****
*/
VOID uniform_dep(
    INT      *p0,
    INT      *P,
    INT      *q0,
    INT      *A,
    INT      *a0,
    INT      *b0,
    INT      m,
    INT      n,
    INT      *exist,
    FILE    *file_ptr
)
{

```

```

INT      *U, *S, *T, *DP;
INT      *TS, *D, *C, *DIR;
INT      i,j,k,a,M,11,nr;
INT      temp_sum;
INT      found, rho, level;
INT      mint, maxt;
BOOL     lex_pos, lex_neg, equal;

U = (INT *) calloc(m*m, sizeof(INT));
S = (INT *) calloc(m*n, sizeof(INT));
T = (INT *) calloc(m, sizeof(INT));
C = (INT *) calloc(n, sizeof(INT));
DP= (INT *) calloc(m*m, sizeof(INT));
D = (INT *) calloc(m, sizeof(INT));
DIR = (INT *) calloc(m, sizeof(INT));

/***
*** STEP 1 of Algorithm 5.2

*** Call Diophantine Equation Solver (1) to Solve
*** DA = a0 - b0
*/
FOR(i = 0; i < n; i++)
  C[i] = a0[i] - b0[i];
diophantine_1(A, U, S, C, T, m, n, exist);

/***
*** STEPS 2 & 3 & 4 of Algorithm 5.2
*** Find rho and M.
*** IF no integer solution, no dependence, Return!
*** (USE D = tU )
*/
IF(*exist == -1)
  goto free_ret;
rho = *exist;
FOR(j = 0; j < m; j++) {
  D[j] = 0;
}

```

```

FOR (i = 0; i < rho; i++)
    D[j] += T[i] * U[i *m +j];
}
M = m - rho;

/**
*** STEP 5 of Algorithm 5.2
***
*** If M = 0 or 1, the system of inequalities
***      p0 - q0 <= t(UP) <= q0 - p0
*** is easy to solve. Else, we need Fourier's
*** elimination method for solving the problem.
*** We assume dependence in that case.
*** */
}

IF(M >= 2) {
    *exist = 0;
    fprintf(file_ptr,
        " Possible Dependence!\n");
    goto free_ret;
}

maxt = MAXINT;
mint = -MAXINT;
FOR(j = 0; j < m; j++) {
    DP[j] = 0;
    FOR(k = 0; k < m; k++) {
        DP[j] += D[k] * P[k*m + j];
    }
}
FOR(i = 0 ; i < m; i++) {
    IF( M == 0 &&
        ((p0[i]-q0[i]) > DP[i]) ||
        (q0[i]-p0[i]) < DP[i])) {
        *exist = -1;
        goto free_ret;
    }
    IF( M != 0) {
        a =0;
    }
}

```

```

FOR(k = 0; k < m; k++)
    a += U[(m-1)*m + k] * P[k*m + i];
IF( a > 0) {
    mint = MAX(mint,
        ((p0[i]-q0[i]-DP[i])/a));
    maxt = MIN(maxt,
        ((q0[i]-p0[i]-DP[i])/a));
}
IF(a < 0) {
    mint = MAX(mint,
        ((q0[i]-p0[i]-DP[i])/a));
    maxt = MIN(maxt,
        ((p0[i]-q0[i]-DP[i])/a));
}
}

/**
 ***
*** STEPS 6 and 7 of Algorithm 5.2
***/
IF(M == 0) {
    lex_pos = FALSE;
    lex_neg = TRUE;
    FOR(i = 0; i < m; i++) {
        IF(D[i] > 0) {
            BREAK;
        }
        IF(D[i] < 0) {
            lex_neg = FALSE;
            lex_pos = TRUE;
            BREAK;
        }
    }
    IF(lex_pos) {
        FOR(i = 0; i < m; i++) {
            D[i] = -D[i];
        }
    }
}
ELSE {
    lex_pos = FALSE;
    lex_neg = FALSE;
}

```

```

FOR(i = mint; i <= maxt; i++) {
    equal = TRUE;
    FOR(j = 0; j < m; j++) {
        a = D[j] + (U[(m-1)*m + j] * i);
        IF(a > 0) {
            lex_neg = TRUE;
            equal = FALSE;
            BREAK;
        }
        IF(a < 0) {
            lex_pos = TRUE;
            equal = FALSE;
            BREAK;
        }
    }
    IF(equal)
        lex_neg = TRUE;
    IF(lex_neg && lex_pos) BREAK;
}

}

IF(lex_pos){
    fprintf(file_ptr,
        " Dependence of S on T is nonempty.\n");
}
IF(lex_neg){
    fprintf(file_ptr,
        " Dependence of T on S is nonempty.\n");
}
/***
*** STEP 8 of Algorithm 5.2
*** Find DIR and distance vector D and Level.
*** */
level = -1;
IF(M == 0) {
    fprintf(file_ptr,
        " The components of the D vector are:");
    FOR(i = 0; i < m; i++){
        DIR[i] = SIG(D[i]);
        fprintf(file_ptr, " %d ",D[i]);
        IF(level == -1 && D[i] )
}

```

```
        level = i+1;
    }
    fprintf(file_ptr, "\n");
    fprintf(file_ptr,
            " The Level of Dependence is:");
    fprintf(file_ptr, " %d \n",level);
}
ELSE{
    FOR(i = mint; i <= maxt; i++) {
        fprintf(file_ptr,
                " The components of the D vector for t = %d are:",i);
        FOR(j = 0; j < m; j++) {
            a = D[j] + (U[(m -1) * m + j] * i);
            fprintf(file_ptr, " %d ",a);
            DIR[j] = SIG(a);
            IF(level == -1 && a )
                level = j+1;
        }
        fprintf(file_ptr, "\n");
    }
    fprintf(file_ptr,
            " The Level of Dependence is:");
    fprintf(file_ptr, " %d \n",level);
}

}
/***
***  

*** STEP 9 of Algorithm 5.2  

***  

****/  

free_ret:  

    free((CHAR *) U);
    free((CHAR *) S);
    free((CHAR *) T);
    free((CHAR *) D);
    free((CHAR *) C);
    free((CHAR *) DP);
    free((CHAR *) DIR);
    RETURN;
}  

****  

****  

***** This is the routine to solve the dependence equation
```

```

***** in a rectangular loop ( $P = Q = I_m$ ) with
*****      A and B as diagonal matrices.
*****
***** Input: The m vectors  $p_0$  and  $q_0$ ,
*****      m X m diagonal matrix A,
*****      m X m diagonal matrix B,
*****      m vectors  $a_0$  and  $b_0$ ,
*****      the integer m and a pointer to
*****          the result flag,
*****      a lexicographically positive direction
*****      vector dir
*****
***** Output: The solution to the dependence equation.
*****
***** Steps: Follows the book very closely.
*****
****/

```

```
VOID diagonal_dep(
```

```

INT      *p0,
INT      *q0,
INT      *A,
INT      *a0,
INT      *B,
INT      *b0,
INT      m,
INT      *exist,
FILE     *file_ptr,
INT      *dir
)
{
    INT      *U, *S, *T, *c;
    Psi      *s, *s0, *s1, *s_1, *pos_s, *neg_s;
    INT      *min1, *max1, *min_1, *max_1;
    INT      i, j, k, r, a, N;
    BOOL     pos, neg, zero;

    U      = (INT *) calloc(m*m, sizeof(INT));
    S      = (INT *) calloc(m*m, sizeof(INT));
    T      = (INT *) calloc(m, sizeof(INT));
    c      = (INT *) calloc(m, sizeof(INT));
    min1  = (INT *) calloc(m, sizeof(INT));
    max1  = (INT *) calloc(m, sizeof(INT));
    min_1 = (INT *) calloc(m, sizeof(INT));

```

```

max_1 = (INT *) calloc(m, sizeof(INT));
s    = (Psi *)calloc(m, sizeof(Psi));
s0   = (Psi *)calloc(m, sizeof(Psi));
s1   = (Psi *)calloc(m, sizeof(Psi));
s_1  = (Psi *)calloc(m, sizeof(Psi));
pos_s = (Psi *)calloc(m, sizeof(Psi));
neg_s = (Psi *)calloc(m, sizeof(Psi));

/**
*** STEP 1 of Algorithm 5.3
***
*** Find (c1, c2, ... ,cm) <- (b10-a10, b20-a20,...)
***
**/

FOR(i = 0; i < m; i++)
  c[i] = b0[i] - a0[i];

/**/

*** STEP 2 of Algorithm 5.3
***
*** For r = 1, 2, ... m call Algorithm 3.1
*** to find the set Psi(r) of all solutions (ir, jr)
*** to the diophantine equation
***
***     arr*ir - brr * jr = cr;
*** s.t. pr <= ir <= qr and pr <= jr <= qr
***
**/

FOR( r = 0; r < m; r++) {
  two_var(A[r*m + r], B[r*m + r], c[r], p0[r],
          q0[r], s+r, s0+r, s1+r, s_1+r,
          min1+r, max1+r, min_1+r, max_1+r);
}

/**/
*** STEP 3 of Algorithm 5.3
***
*** If s[r] is NULL for any r, there is no dependence!
**/

FOR( r = 0; r < m; r++) {
  IF((s+r)->empty_set) {

```

```

        *exist = -1;
        goto free_return;
    }
}

/***
*** STEP 4 of Algorithm 5.3
***
*** If the input direction vector is
*** lexicographically positive, dependence of
*** T on S with dir and S on T with dir is
*** computed and printed.
*/
pos = FALSE;
neg = FALSE;
FOR( r = 0; r < m; r ++ ) {
    IF(dir[r] == 0) continue;
    IF(dir[r] > 0) {
        pos = TRUE;
        neg = FALSE;

        BREAK;
    }
    IF(dir[r] < 0) {
        pos = FALSE;
        neg = TRUE;
        BREAK;
    }
}
IF(pos) {
    pos = TRUE;
    neg = TRUE;
    FOR(r = 0; r < m; r++) {
        IF(dir[r] > 0) {
            IF((s1+r) ->empty_set) {
                IF(pos)
                    fprintf(file_ptr,
                            " No Dependence of T on S\n");
                pos = FALSE;
            }
            ELSE {
                pos_s[r] = s1[r];
            }
            IF((s_1+r)->empty_set) {
                IF(neg)
                    fprintf(file_ptr,

```

```

        " No Dependence of S on T\n");
        neg = FALSE;
    }
    ELSE{
        neg_s[r] = s_1[r];
    }
}
IF(dir[r] < 0) {
    IF((s_1+r) ->empty_set) {
        IF(pos)
            fprintf(file_ptr,
                    " No Dependence of T on S\n");
        pos = FALSE;
    }
    ELSE{
        pos_s[r] = s_1[r];
    }
    IF((s1+r)->empty_set) {
        IF(neg)

            fprintf(file_ptr,
                    " No Dependence of S on T\n");
        neg = FALSE;
    }
    ELSE{
        neg_s[r] = s1[r];
    }
}
IF(dir[r] == 0) {
    IF((s0+r) ->empty_set) {
        IF(pos)
            fprintf(file_ptr,
                    " No Dependence of T on S\n");
        pos = FALSE;
        IF(neg)
            fprintf(file_ptr,
                    " No Dependence of S on T\n");
        neg = FALSE;
    }
    ELSE{
        pos_s[r] = s0[r];
        neg_s[r] = s0[r];
    }
}
IF(!(pos || neg)) BREAK;

```

```

}

IF(pos) {
    fprintf(file_ptr, " Dependence of T on S:\n");
    fprintf(file_ptr, "(");
    FOR(r = 0; r < m; r++) {
        fprintf(file_ptr, "%d + %d t%d ",
            ((pos_s+r)->y.a_part - (pos_s+r)->x.a_part),
            ((pos_s+r)->y.b_part - (pos_s+r)->x.b_part),
            r+1);
        IF(r != m-1)fprintf(file_ptr, ", ");
    }
    fprintf(file_ptr, ")");
    FOR(r = 0; r < m; r++) {
        fprintf(file_ptr, "%d <= t%d <= %d ",
            (pos_s+r)->lbnd_t,r+1,(pos_s+r)->ubnd_t);
        IF(r != m-1)fprintf(file_ptr, ", ");
    }
    fprintf(file_ptr, ")\n");
}

IF(neg) {
    fprintf(file_ptr, " Dependence of S on T:\n");
    fprintf(file_ptr, "(");
    FOR(r = 0; r < m; r++) {
        fprintf(file_ptr, "%d + %d t%d ",
            ((neg_s+r)->y.a_part - (neg_s+r)->x.a_part),
            ((neg_s+r)->y.b_part - (neg_s+r)->x.b_part),
            r+1);
        IF(r != m-1)fprintf(file_ptr, ", ");
    }
    fprintf(file_ptr, ")");
    FOR(r = 0; r < m; r++) {
        fprintf(file_ptr, "%d <= t%d <= %d ",
            (neg_s+r)->lbnd_t,r+1,(neg_s+r)->ubnd_t);
        IF(r != m-1)fprintf(file_ptr, ", ");
    }
    fprintf(file_ptr, ")\n");
}
}

ELSE {
    fprintf(file_ptr,
        " Lexicographically Nonpositive \
        Direction Vector!\n");
}
/**
```

```
*** STEP 5 of Algorithm 5.3
*/
zero = TRUE;
FOR( r = 0; r < m; r ++ ) {
    IF((s0+r)->empty_set) {
        zero = FALSE;
        BREAK;
    }
}
IF(zero){
    fprintf(file_ptr,
    " There is a dependence of T on S with 0\n");
}
/***
*** STEP 6 of Algorithm 5.3
*/
IF(!(pos || neg || zero)) {
    *exist = -1;
    goto free_return;

}
*exist = 0;
/***
*** STEP 7 of Algorithm 5.3
*/
free_return:
    free((CHAR *) U);
    free((CHAR *) S);
    free((CHAR *) T);
    free((CHAR *) c);
    free((CHAR *) min1);
    free((CHAR *) max1);
    free((CHAR *) min_1);
    free((CHAR *) max_1);
    free((CHAR *) s);
    free((CHAR *) s0);
    free((CHAR *) s1);
    free((CHAR *) s_1);
    free((CHAR *) pos_s);
    free((CHAR *) neg_s);
    RETURN;
}
```

Output From Sample Runs Of Linear Dependence Algorithm

(1)

THE MATRIX A is :

2	0
0	5

THE VECTOR a0 is : 3 -1

THE MATRIX B is :

2	0
0	5

THE VECTOR b0 is : -1 -6

THE MATRIX P is :

1	0
0	1

THE VECTOR p0 is : 10 7

THE MATRIX Q is :

1	0
0	1

THE VECTOR q0 is : 200 167

Uniform Dependence Testing
 Dependence of T on S is nonempty.
 The components of the D vector are: 2 1
 The Level of Dependence is: 1

(2)

THE MATRIX A is :

2	3
3	4

THE VECTOR a0 is : -2 1

THE MATRIX B is :

2	3
3	4

THE VECTOR b0 is : 1 3

THE MATRIX P is :

1	0
0	1

THE VECTOR p0 is : 1 71

THE MATRIX Q is :

1 0
0 1

THE VECTOR q0 is : 1000 300

Uniform Dependence Testing

Dependence of T on S is nonempty.

The components of the D vector are: 6 -5

The Level of Dependence is: 1

(3)

THE MATRIX A is :

2
3

THE VECTOR a0 is : 12

THE MATRIX B is :

2
3

THE VECTOR b0 is : -5

THE MATRIX P is :

1 -1
0 1

THE VECTOR p0 is : 0 0

THE MATRIX Q is :

1 -1
0 1

THE VECTOR q0 is : 100 50

Uniform Dependence Testing

Dependence of S on T is nonempty.

Dependence of T on S is nonempty.

The components of the D vector for t = -3 are: -26 23

The components of the D vector for t = -2 are: -23 21

The components of the D vector for t = -1 are: -20 19

The components of the D vector for t = 0 are: -17 17

The components of the D vector for t = 1 are: -14 15

The components of the D vector for t = 2 are: -11 13

The components of the D vector for t = 3 are: -8 11

The components of the D vector for t = 4 are: -5 9

The components of the D vector for t = 5 are: -2 7

The components of the D vector for t = 6 are: 1 5

The components of the D vector for t = 7 are: 4 3

The components of the D vector for t = 8 are: 7 1

The components of the D vector for $t = 9$ are: 10 -1
 The components of the D vector for $t = 10$ are: 13 -3
 The components of the D vector for $t = 11$ are: 16 -5
 The components of the D vector for $t = 12$ are: 19 -7
 The components of the D vector for $t = 13$ are: 22 -9
 The components of the D vector for $t = 14$ are: 25 -11
 The components of the D vector for $t = 15$ are: 28 -13
 The components of the D vector for $t = 16$ are: 31 -15
 The Level of Dependence is: 1

(4)

THE MATRIX A is :

2 1

THE VECTOR a0 is : -2 3

THE MATRIX B is :

-1 2

THE VECTOR b0 is : 300 9

THE MATRIX P is :

1

THE VECTOR p0 is : 10

THE MATRIX Q is :

1

THE VECTOR q0 is : 200

General Dependence Testing

Dependence of S on T is nonempty.

The components of the D vector are: 64

(5)

THE MATRIX A is :

2 0 0

0 5 1

THE VECTOR a0 is : 3 -1 0

THE MATRIX B is :

1 2 0

0 0 3

THE VECTOR b0 is : -1 -6 2

THE MATRIX P is :

1 0

0 1

THE VECTOR p0 is : 10 7

THE MATRIX Q is :

1	0
0	1

THE VECTOR q0 is : 200 167

General Dependence Testing

Dependence of T on S is nonempty.

The components of the D vector for t = 2 are: 32 -16

The components of the D vector for t = 3 are: 47 -24

The components of the D vector for t = 4 are: 62 -32

The components of the D vector for t = 5 are: 77 -40

The components of the D vector for t = 6 are: 92 -48

(6)

THE MATRIX A is :

6	0
0	2

THE VECTOR a0 is : -1 -1

THE MATRIX B is :

4	0
0	2

THE VECTOR b0 is : 9 9

THE MATRIX P is :

1	0
0	1

THE VECTOR p0 is : 0 0

THE MATRIX Q is :

1	0
0	1

THE VECTOR q0 is : 35 35

Diagonal Dependence Testing

The input direction vector is: 1 -1

No Dependence of S on T

Dependence of T on S:

((t1 , -5): 1 <= t1 <= 10 , 5 <= t2 <= 35)

(7)

THE MATRIX A is :

6	0
0	2

THE VECTOR a0 is : -1 -1

THE MATRIX B is :

4 0
0 2

THE VECTOR b0 is : 9 9

THE MATRIX P is :

1 0
0 1

THE VECTOR p0 is : 0 0

THE MATRIX Q is :

1 0
0 1

THE VECTOR q0 is : 35 35

Diagonal Dependence Testing

The input direction vector is: 1 1

No Dependence of T on S

Dependence of S on T:

((t1 , -5): -1 <= t1 <= -1 , 5 <= t2 <= 35)

Bibliography

- [AlKe84] John R. Allen & Ken Kennedy. Automatic Loop Interchange. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 17-22, 1984. pp. 233-246. Available as *SIGPLAN Notices*, vol. 19, no. 6. June 1984.
- [AlKe87] J. R. Allen & K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, pp. 491-542. October 1987.
- [Alle83] John R. Allen. Dependence Analysis for Subscripted Variables and its Application to Program Transformations. *PhD Thesis*. Dept. of Mathematical Sciences, Rice University, Houston, Texas. April 1983. Available as *Document 83-14916* from University Microfilms, Ann Arbor, Michigan.
- [Bane76] Utpal Banerjee. Data Dependence in Ordinary Programs. *MS Thesis. Report 76-837*. Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois. November 1976.
- [Bane79] Utpal Banerjee. Speedup of Ordinary Programs. *PhD Thesis. Report 79-989*. Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois. October 1979. Available as *Document 80-08967* from University Microfilms, Ann Arbor, Michigan.
- [Bane88a] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, Massachusetts. 1988.

- [Bane88b] Utpal Banerjee. An Introduction to a Formal Theory of Dependence Analysis. *The Journal of Supercomputing*, vol. **2**, no. 2, pp. 133–149. October 1988.
- [Bane90] Utpal Banerjee. A Theory of Loop Permutations. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, Urbana, Illinois, August 1989. Available as *Languages and Compilers for Parallel Computing* (eds. D. Gelernter, A. Nicolau & D. Padua). pp. 54–74. The MIT Press, Cambridge, Massachusetts. 1990.
- [Bane91] Utpal Banerjee. Unimodular Transformations of Double Loops. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, California, August 1–3, 1990. Available as *Advances in Languages and Compilers for Parallel Computing* (eds. A. Nicolau, D. Gelernter, T. Gross & D. Padua). pp. 192–219. The MIT Press, Cambridge, Massachusetts. 1991.
- [BCKT79] Utpal Banerjee, Shyh-Ching Chen, David J. Kuck & Ross A. Towle. Time and Parallel Processor Bounds for Fortran-Like Loops. *IEEE Transactions on Computers*, vol. **C-28**, no. 9, pp. 660–670. September 1979.
- [BENP93] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau & David A. Padua. Automatic Program Parallelization. To appear in *IEEE Proceedings* in 1993.
- [BuCy86] M. Burke & R. Cytron. Interprocedural Dependence Analysis and Parallelization. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, California, June 25–27, 1986. pp. 162–175. Available as *SIGPLAN Notices*, vol. **21**, no. 7. July 1986.
- [Coha73] W. Cohagan. Vector Optimization for the ASC. *Proceedings of the 7th Annual Princeton Conference on Information Sciences and Systems*, Princeton, New Jersey, March 22–23, 1973. pp. 169–174. Princeton University Press, Princeton, New Jersey. 1973.

- [D'Hol89] Erik H. D'Hollander. Partitioning and Labeling of Index Sets in Do Loops with Constant Dependence Vectors. *Proceedings of the 1989 International Conference on Parallel Processing, vol. II: Software*, St. Charles, Illinois, August 8–12, 1989. pp. 139–144. Penn. State University Press, University Park, Pennsylvania. August 1989.
- [D'Hol92] Erik H. D'Hollander. Partitioning and Labeling of Loops by Unimodular Transformations. *IEEE Transactions on Parallel and Distributed Systems*, vol. **3**, no. 4, pp. 465–476. July 1992.
- [Dowl90] Michael L. Dowling. Optimal Code Parallelization using Unimodular Transformations. *Parallel Computing*, vol. **16**, pp. 157–171, 1990.
- [Duff74] R. J. Duffin. On Fourier's Analysis of Linear Inequality Systems. *Mathematical Programming Study 1*, pp. 71–95. North Holland. 1974.
- [Even79] Shimon Even. *Graph Algorithms*. Computer Science Press, Rockville, Maryland. 1979.
- [Gibb85] Alan Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, New York, New York. 1985.
- [GoKT91] Gina Goff, Ken Kennedy & Chau-Wen Tseng. Practical Dependence Testing. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 26–28, 1991. Available as *SIGPLAN Notices*, vol. **26**, no. 6, pp. 15–29. June 1991.
- [Halm65] Paul R. Halmos. *Naive Set Theory*. D. Van Nostrand, Princeton, New Jersey. 1965.
- [HaPo90] Mohammad Haghagh & Constantine D. Polychronopoulos. Symbolic Dependence Analysis for High-Performance Parallelizing Compilers. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, California, August 1–3, 1990. Available as *Advances in Languages and Compilers for Parallel Computing* (eds.

- A. Nicolau, D. Gelernter, T. Gross & D. Padua). The MIT Press, Cambridge, Massachusetts. pp. 310–330. 1991.
- [IrTr89] François Irigoin & Rémi Triolet. Dependence Approximation and Global Parallel Code Generation for Nested Loops. In *Parallel and Distributed Algorithms*, (eds. M. Cosnard et al.). Elsevier (North-Holland), New York, New York. pp. 297–308. 1989.
- [KeMc90] Ken Kennedy & Kathryn S. McKinley. Loop Distribution with Arbitrary Control Flow. *Proceedings of Supercomputing '90*, New York, New York, November 1990. pp. 407-416. IEEE Computer Society Press, Los Alamitos, California.
- [Kert81] S. Kertzner. The Linear Diophantine Equation. *American Mathematical Monthly*, vol. **88**, no. 3, pp. 200–203. March 1981.
- [Knut73] Donald E. Knuth. *The Art of Computer Programming, Vol 1: Fundamental Algorithms*, Second Edition. Addison-Wesley, Reading, Massachusetts. 1973.
- [Knut81] Donald E. Knuth. *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*, Second Edition. Addison-Wesley, Reading, Massachusetts. 1981.
- [KoKP90] Xiangyun Kong, David Klapholz & Kleanthis Psarris. The I Test: A New Test for Subscript Data Dependence. *Proceedings of the 1990 International Conference on Parallel Processing, vol. II: Software*, St. Charles, Illinois, August 13–17, 1990. pp. 204–211. Penn. State University Press, University Park, Pennsylvania. August 1990.
- [Lamp74] L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, vol. **17**, no. 2, pp. 83–93. February, 1974.
- [LiYe89] Zhiyuan Li & Pen-Chung Yew. Some Results on Exact Data Dependence Analysis. *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, Urbana, Illinois, August 1989. Available as *Languages and*

- Compilers for Parallel Computing* (eds. D. Gelernter, A. Nicolau & D. Padua). The MIT Press, Cambridge, Massachusetts. pp. 374–401. 1990.
- [LiYZ90] Zhiyuan Li, Pen-Chung Yew & Chuan-Qi Zhu. An Efficient Data Dependence Analysis for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, vol. **1**, no. 1, pp. 26–34. January 1990.
- [MaHL91] Dror E. Maydan, John L. Hennessy & Monica S. Lam. Efficient and Exact Data Dependence Analysis. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 26–28, 1991. Available as *SIGPLAN Notices*, vol. **26**, no. 6, pp. 1–14. June 1991.
- [Mirs71] L. Mirsky. A Dual of Dilworth's Decomposition Theorem. *The American Mathematical Monthly*, vol. **78**, no. 8, 876–877. October 1971.
- [Mura71] Yoichi Muraoka. Parallelism Exposure and Exploitation in Programs. *PhD Thesis. Report 71-424*. Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois. February 1971.
- [Padu79] David A. Padua. Multiprocessors: Discussion of some Theoretical and Practical Problems. *PhD Thesis. Report 79-990*. Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois. November 1979.
- [PaWo86] David A. Padua & Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, vol. **29**, no. 12, pp. 1184–1201. December 1986.
- [PeCy89] J.-K. Peir and R. Cytron. Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors. *IEEE Transactions on Computers*, vol. **38**, no. 8, pp. 1203–1211. August 1989.
- [Prat76] R. E. Prather. *Discrete Mathematical Structures for Computer Science*. Houghton Mifflin Co., Boston, Massachusetts. 1976.

- [Pugh91] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Proceedings of Supercomputing '91*, Albuquerque, New Mexico, November 1991. IEEE Computer Society Press, Los Alamitos, California.
- [Schr87] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, New York. 1987.
- [ShFo88] Weijia Shang and Jose A. B. Fortes. Independent Partitioning of Algorithms with Uniform Dependences. *Proceedings of the 1988 International Conference on Parallel Processing, vol. II: Software*, St. Charles, Illinois, August 15–19, 1988. pp. 26–33. Penn. State University Press, University Park, Pennsylvania. August 1988.
- [ShFo91] Weijia Shang and Jose A. B. Fortes. Time Optimal Linear Schedules for Algorithms with Uniform Dependences. *IEEE Transactions on Computers*, vol. 40, no. 6, pp. 723–742. June 1991.
- [ShLY89] Z. Shen, Zhiyuan Li & Pen-Chung Yew. An Empirical Study on Array Subscripts and Data Dependencies. *Proceedings of the 1989 International Conference on Parallel Processing, vol. II: Software*, pp. 145–152. The Pennsylvania State University Press, University Park, Pennsylvania. August 1989.
- [Tarj72] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, vol. 1, pp. 146–160. 1972.
- [Towl76] R. Towle. Control and Data Dependence for Program Transformations. *PhD Thesis. Report 76-788*. Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois. March 1976.
- [Wall88] David R. Wallace. Dependence of Multi-Dimensional Array References. *Proceedings of the 1988 International Conference on Supercomputing*, St. Malo, France, July 1988. pp. 418–428. The ACM Press, New York, New York.

- [Will86] H. P. Williams. Fourier's Method of Linear Programming and its Dual. *The American Mathematical Monthly*, vol. **93**, no. 9, pp. 681–695. November 1986.
- [WoBa87] Michael J. Wolfe & Utpal Banerjee. Data Dependence and its Application to Parallel Processing. *International Journal of Parallel Programming*, vol. **16**, no. 2, pp. 137–178. April 1987.
- [WoLa91a] Michael E. Wolf & Monica S. Lam. An Algorithmic Approach to Compound Loop Transformations. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, California, August 1–3, 1990. Available as *Advances in Languages and Compilers for Parallel Computing* (eds. A. Nicolau, D. Gelernter, T. Gross & D. Padua). pp. 243–259. The MIT Press, Cambridge, Massachusetts. 1991.
- [WoLa91b] Michael E. Wolf & Monica S. Lam. A Loop Transformation Theory and an Algorithm to maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, vol. **2**, no. 4, pp. 452–471. October 1991.
- [Wolf82] Michael J. Wolfe. Optimizing Compilers for Supercomputers. *PhD Thesis, Report 82-1105*, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois. October 1982. Available as Center for Supercomputing Research & Development *Report 329*, and as *Document 83-03027* from University Microfilms, Ann Arbor, Michigan.
- [Wolf86a] Michael J. Wolfe. Loop Skewing: The Wavefront Method Revisited. *International Journal of Parallel Programming*, vol. **15**, no. 4, pp. 279–293. August 1986.
- [Wolf86b] Michael J. Wolfe. Advanced Loop Interchanging. *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, Illinois, August 19–22, 1986. pp. 536–543. IEEE Computer Society Press, Los Angeles, California. 1986.

- [Wolf89] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, Massachusetts. 1989.
- [WoTs92] Michael Wolfe & Chau-Wen Tseng. The Power Test for Data Dependence. *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 5, pp. 591–601. September 1992.