UNCLASSIFIED

AD NUMBER
AD224381
LIMITATION CHANGES
TO: Approved for public release; distribution is unlimited.
FROM: Distribution authorized to U.S. Gov't. agencies and their contractors; Administrative/Operational Use; 14 MAR 1956. Other requests shall be referred to Department of Defense, Public Affairs Office, Washington, DC 20301.
AUTHORITY
Rand ltr, 2 Feb 1967

THIS PAGE IS UNCLASSIFIED

UNCLASSIFIED

AD 224381

DEFENSE DOCUMENTATION CENTER

FOR

SCIENTIFIC AND TECHNICAL INFORMATION

CAMERON STATION, ALEXANDRIA, VIRGINIA

UNCLASSIFIED

NOTICE: When government or other drawings, specifications or other data are used for any purpose other than in connection with a definitely related government procurement operation, the U. S. Government thereby incurs no responsibility, nor any obligation whatsoever; and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use or sell any patented invention that may in any way be related thereto.

UNANNOUNCED

AD No. 224 381
DDC FILE COPY

DDC
MAR 26 1964

(5) 739200 RAND Corp., Santa Monica, Calif.

EVOLUTION OF COMPUTER CODES FOR LINEAR PROGRAMMING

By William Orchard-Hays.

(14) Rept. P-810

14 March 1956

The RAND Corporation
1700 MAIN ST. • SANTA MONICA • CALIFORNIA

FOREWORD

In late fall of 1952, Dr. George B. Dantzig at The RAND Corporation requested that a programmer be assigned to work with him in developing machine computing procedures for the simplex method. The writer was given the assignment, which has since grown into a full-time job, and has been assisted, since late 1953, by Miss Leola Cutler and others of RAND's programming staff in both developing codes and running problems. The machine used in the beginning was the Model II CPC with five 941 units. The limitations of this equipment forced changes in methodology which later proved applicable to codes for the IBM 701. The relative power of the final 701 set-up aroused the interest of IBM's Data Processing Center and, with the advent of the 704, RAND and IBM decided to develop a linear programming system for this machine jointly. Mr. Hal Judd of IBM has worked with Miss Cutler and the writer on this project and much of the material below was originally prepared to describe the 704 system. The ideas here presented are not original to the 704 codes, however, and the methods are not tied to any one machine, except as noted. Applications to other machines are discussed briefly in Part C.

INTRODUCTION

The growing literature on the subject of linear programming falls roughly into three categories:* (i) applications of the linear programming technique to problems of the military, industry and business, and occasionally to problems simply of academic interest; (ii) methods for solving the central mathematical problem posed by the resulting models; and (iii) computing techniques for implementing these methods. Our concern is mainly with the last although it is impossible to completely separate these three aspects. A concept of analysis has value only if it shows promise of being applicable to real-life problems. Where calculations are involved, they must at least be tractable in the abstract and feasible within current computational capabilities.

It is not our purpose to attempt either a definition of linear programming and its uses or a comparison of the basically different methods of calculation. It is now pretty generally agreed that the simplex method of Dr. George Dantzig provides the best known approach. We must insist, however, that it is the fundamental theorems of this method which are of importance and not the particular algorithm and tableau which were set down in the original exposition. The calculations can be carried out in different ways and adaptations and extensions are easy to devise, at least in theory. One of the earliest and most widely used special cases - the transportation problem -- was sufficiently important to warrant development of efficient computer codes for this one purpose, but it is difficult to find other sub-problems as well defined or as easily simplified. Consequently, a general algorithm has been applied in nearly all other cases but both theory and experience indicate that there is an upper bound well below 500 to the number of restraints that can be

---

* In lieu of references in the text, a catalogued bibliography is given at the end of the paper.

handled efficiently in this manner with foreseeable machines. Though this figure is approximate, it is probably already too high for some problems. This has been the cause of much concern. There has been a continuous effort at RAND for three years to develop general codes for larger and larger problems. Being now able to handle some 250 restraint equations in any number of variables and not yet having satisfied the demands of many realistic problems, we conclude that further attempts in this direction will be fruitless. We feel, however, that the present codes are of considerable value and, more importantly, that their evolution has shown the direction which future developments must take. In this regard, perhaps the most important result is the mutual realization by those who formulate problems and methods and those who devise computer programs that, for better or worse, they are joined together.

Our aim is to discuss the growth of these codes, from the standpoint of both mathematical and coding techniques, and then to give a brief resume of current and proposed developments in this field at RAND. The presentation will be in three main parts which will bear, more or less respectively, on the following three theses.

A. In order to handle efficiently a large number of problems of one type, codes must be designed with an eye to the two sides of computational problems: (1) the method must be adapted to the characteristics of the machine, and (2) provision must be made for handling easily any reasonable special requirements, for making a variety of decisions automatically and for easy operation.

B. In order to attain the required efficiency and flexibility, the codes must themselves be organized in a manner analagous to the method and not based on standard routines or universal methods.

C. For solving very large problems at reasonable cost, it will be necessary to develop special techniques for special classes.

PART A - THE REVISED SIMPLEX METHOD WITH SPECIAL FEATURES

A general familiarity with the simplex method will be assumed. For example, no proofs will be given for the standard theorems concerning basic solutions or the simplex criterion. However, the algorithm using the product form of the inverse and certain other features will be developed in detail. The Part is divided into twelve sections. Sections X and XII contain descriptions of devices not previously written up in detail but they have been in use for some time and their usefulness is established.

I - EARLY CPC SET-UPS AT RAND

Since it was felt that the transformation of the entire tableau in the original simplex method caused unnecessary work and a waste of storage capacity, especially in problems with many more variables than restraints, Dantzig proposed that the CPC be set up for the revised simplex method. Here only the inverse of the basis is maintained, together with the transformed right hand side and a pricing vector which form an extra column and row. Since examples of "cycling" due to degeneracy are extremely rare and artificial, no provision was to be made for rigorously resolving this matter. Previous experience with hand calculations and on the SEAC had indicated that such a decision caused no practical difficulty.

It was realized that the CPC was an inadequate machine for linear programming but it was thought that it would provide a good training and testing ground during the interim until large machines were available. This was borne out by subsequent experience.

After some months of study, planning and board wiring, a set-up was in operation which would handle 27 restraints in 70 variables. Operation was fairly automatic, it being necessary to feed through two decks of cards for

each iteration. But it was painfully slow, due mainly to the punching of a new inverse deck on each cycle. Watching this operation one time was enough to convince anyone that the whole set-up should be junked.

Although most elements of the inverse were zero during the first few iterations, the number of non-zero elements increased on succeeding cycles. It was awkward to delete zeros since the necessary bookkeeping really demanded a stored program machine. However, even on a large machine, a collapsed inverse matrix requires excessive bookkeeping and searching if both the inverse and its transpose are needed, that is, if sometimes one wishes combinations of rows and sometimes combinations of columns. Although this could be avoided in the straight simplex method, it is the kind of flexibility that is needed for variations.

Indeed, another shortcoming of the set-up was the very fact that the CPC had been made into a "black box" which rendered it inflexible and hence impractical in use. Model-makers, like women, reserve the right to change their minds.

Dantzig recalled an old suggestion of Alex Orden on the product form of inverse and upon investigation this approach turned out to resolve most of the difficulties then current. A new CPC set-up was designed which could handle 40 restraints in 99 variables (or more with a little care) and was used successfully on several problems. Among these was a series of models in a study of petroleum refinery scheduling being conducted by Dr. Alan S. Manne and economic processing models in general being conducted by Dr. Harry Markowitz. More work was imposed on the machine operator but considerable flexibility was attained. Based on work of Manne, Dantzig, Markowitz and the writer, techniques were arranged to vary the right hand side, the optimizing form and single elements of the matrix. These are described in later sections.

II - THE RAND SIMPLEX CODES FOR THE 701

In mid-1953 work was begun on a 701 code to employ the general features of the second CPC set-up. As many are well aware, preparation for the 701 was a big step from the CPC, regardless of the problems worked on. Many mistakes were made and later corrected as sophistication was gained. However, the writer fell into one fallacy which is hard to explain, let alone justify.

The CPC set-ups had used 8-digit floating decimal arithmetic. As many as 80 iterations on 35-order systems had been performed before accumulated round-off error became troublesome. The initial goal for the 701 was 100 restraint equations and high operating speed. In trying for speed, it was decided to use single-precision, fixed-point arithmetic. The code was tested on a 24-order system consisting mainly of the identity matrix. But when a dense 7-order system was tried, the code stopped after a few iterations with an error indication. After some hand computations, it was realized that the error stop was legitimate, it was actually due to round-off! The code was revised in an attempt to squeeze out more significant places, but to no avail. The basic transformations were simply too inaccurate.

Finally realizing the significance of the CPC experience, we rewrote the transformation sections of the code to perform double-precision, floating-point arithmetic, i.e. carrying nearly 16 decimal digits (52 binary bits) plus a binary exponent. This permitted 100-order systems to be solved and, in fact, 250 equations are now handled with essentially the same arithmetic (54 binary bits plus exponent.) It has been possible to keep running time down on some operations by using fixed-point arithmetic, partly double and partly single precision. We feel that the added complexity is a small price to pay for the results. The inputs are still required to fall into a restricted fixed-point range. This causes no difficulty in linear systems usually of low precision

and is an effective way to obtain reasonably good scaling from a variety of users of the system.

Other more excusable flaws began to appear. The initial flexibility attained on the CPC was discouragingly difficult on the 701. Improvements and changes for easier operation or special requirements involved prodigous tasks of code-checking. Further, some believed that a high speed computer code should read original data cards, solve the problem, print the answers and stop. We tried this mode of operation and found it impractical for large computations with large amounts of input. For reasons well known at any computing center, the probability is practically zero that a large job will go to completion with no error and in one run. There is the added uncertainty as to whether the right problem got into the machine correctly in the first place. We have had the experience of getting correct answers to the wrong problem.

Consequently, the whole program was completely re-organized and divided into separate parts. The initial conversion, editing and packing of data is now done by a separate data assembly code which produces binary cards and magnetic tape. Errors are expected and provisions made beforehand for recovery. The main code is organized into replaceable or interchangeable regions which correspond to the major divisions of the method. The present programs, which are the result of two subsequent revisions, are discussed further in Part B.

III - ADVANTAGES OF THE PRODUCT FORM OF INVERSE

The advantages claimed for the revised simplex method with the product form of inverse will now be summarized and the algorithm then developed in detail. It is necessary to keep in mind the following facts concerning the matrix of coefficients of a typical linear programming problem and the re-

quirements of the simplex method.

(a) The matrix is usually very sparse, the percentage of non-zero elements being typically 5 to 15 per cent of the total.

(b) The given data is almost invariably of low precision and easily scaled.

(c) Some problems have a large ratio of number of variables to number of restraints, between 5 to 1 and 10 to 1 being fairly common. (One problem run on the 701 had a ratio of about 30 to 1.)

(d) The rules of elimination in effecting a change of basis (where two successive bases differ only by one column) are the same no matter what particular algorithm is used. Although such a transformation is easily represented, its total effect on the numerical representation of a set of vectors, i.e. a matrix, may be extensive.

With these observations in mind, the advantages of the present method can be stated as follows.

(1) Since the original data matrix is not transformed from iteration to iteration, it is clear from (a) and (b) that an elaborate organization of the data can be performed at the outset, once and for all, so that it is in the most convenient and compact form for use throughout the problem. Since the original matrix is referred to only once during the cycle, it can be stored in auxiliary storage (say on a tape) out of the way, and its compact form minimizes transfer time as well as computing time.

(2) It is clear from (c) that it would be very expensive in some problems to transform the whole matrix on each cycle. If there are n variables and m constraints and if n is, say, only 5m, it is very unusual for the process to take 5m iterations. In other words,

many vectors (columns) are never selected at all to come into the basis and hence there is no use doing anything to them.

(3) As remarked above, the modification and re-recording of the m^2 elements of a full inverse is awkward and time-consuming, especially if zeros are deleted. On the other hand, the produce form requires the recording of only one additional column (plus an index) on each iteration. These columns will almost surely contain many zeros and may be condensed since they are always applied column-wise and recursively to a single vector.

(4) The product form is extremely amenable to modifications of the method since the inverse and its transpose are equally easy to apply.

One apparent disadvantage of the product form is that, while only one new column is recorded each cycle, it is necessary to read T columns to apply this form where T is the number of iterations already performed. Thus, after m iterations, where m is the number of restraints, one must read more information than in reading a full inverse (not taking condensation into account.) This is still profitable for something over 2m iterations as can be seen as follows.

With an explicit inverse, we must read it twice and write it once each iteration, giving 3mT columns handled on T iterations.

With the product form, we must read t-1 columns twice on iteration t and write one new column, giving

$$\sum_{t=1}^{T}\left[2(t-1) + 1\right] = T(T-1) + T = T^2 \text{ columns handled on T iterations.}$$

Setting $T^2 = 3mT$ gives $T = 3m$. However something must be allowed for the fact that less arithmetic accompanies the writing than the reading. This disadvantage is more apparent than real, however. After, say, 2m iterations, round-off error will begin to be noticeable on large problems no matter

which form is used. Many problems are solved in 2m iterations or less but, if it takes more, one can re-invert the basis matrix, at any time, producing not more than m columns of information. The time for this inversion is much less and the accuracy of the resulting transformations is as good or better than after the same number of full simplex cycles. (The order of elimination in inverting is designed to maintain accuracy.) A special code is provided for this purpose. It is useful for solving any system of linear equations, especially where several right hand sides are used with a sparse matrix.

It is helpful with this form of inversion, to consider the matrix of coefficients of the restraint equations as a collection, or set, of column vectors, ignoring the fortuitous ordering given to the variables in the formulation of the problem. Whatever scrambling of columns that may occur in the process is recorded in a list of basis headings which accompany and identify the basic variables of a particular solution. This point of view will be adopted throughout the present discussion.

IV - NOTATION

In discussions of matrices, their transposes, columns, rows, elements, etc., there has arisen in the literature a hodge-podge of notation - upper- and lower-case, bars, stars, primes, underscoring, varied type faces, etc. Hence we introduce the following simplified and unified notation aimed at reducing the number of symbols and type forms required to talk about matters of inherent simplicity.

Real numbers will be represented by lower-case, light face Latin or Greek letters. However, these letters will be indexed as necessary and an indexed quantity may mean one particular element, a row, a column or a matrix. The context will always make clear which is meant. In exceptional cases, specific

remarks will be made on the side.

The distinction between Latin and Greek letters will appear later. In either case, a row vector will be denoted by a general subscript, where the range of values which this index takes on is defined when it is first mentioned. An element of the row vector will be denoted by a specific subscript.

c_j is a row vector, c_3 is the element of index 3.

A column vector will be denoted by a general superscript, one of its elements by a specific superscript.

b^j is a column vector, b^2 is the element of index 2.

A matrix will be denoted by both a superscript and a subscript.

a^i_j is a matrix

a^5_j is the row consisting of the element of index 5 in each column.

a^i_4 is the column consisting of the element of index 4 in each row.

a^6_2 is the element in row 6 and column 2.

It is important to note that no sifnificance is attached to the letter used for an index. Thus, a_i and a_j refer to the same quantities. However, a^i is the transpose of a_i. The transpose of a matrix must be denoted by defining a new letter, e.g., $b^i_j = a^j_i$ defines b^i_j as the transpose of a^i_j.

We will also adopt a modified form of the summation convention, as follows.

Summation Convention: Whenever the same letter appears twice in a term, as the index of a row and then following as the index of a column, this letter is a dummy index and summation is understood over the (previously defined) range of these indices.

For given coefficients a^i_j and constants b^i, then, a system of m simultaneous linear equations in n unknowns x^j would be written simply

$$a^i_j\, x^j = b^i \qquad (i = 1, \ldots, m;\ \ j = 1, \ldots, n).^*$$

A matrix may be multiplied on the left by a row vector to produce a row vector:

$$c_i\, a^i_j = d_j\ .$$

If a matrix is multiplied on the left by a row and on the right by a column, the result is a scalar:

$$c_i\, a^i_j\, x^j = z\ .$$

Note that $a^i\, b_i$ does <u>not</u> denote summation. If it means anything, it simply denotes a set of products. The quantity

$$c^i_j = a^i\, b_j$$

is a matrix, the so-called outer product of a^i and b_j.

Uppercase letters (except T) will be used for sets. The use of letters as indices is as follows:

h, i, j, k	for general indices,
ℓ, m, n,	for limits,
p, q, r, s	for specific indices.

When it is desired to index a quantity over time, i.e. iterations, an index (denoted in general by t) enclosed in parentheses will be used, e.g. $\alpha^i_{s(t)}$ is produced from $\alpha^i_{s(1)}$ after t-1 transformations. The capital letter T will be used as the current limit for t, that is, T is the current iteration number and $t = 1, \ldots, T$.

<u>V - FORM OF THE PROBLEM AND INITIAL SET-UPS</u>

Since it is mandatory to start with the identity matrix as the initial basis in all circumstances, we define the standard linear programming problem in two forms. Certain deviations from these standard forms are discussed below.

---

* Note the similarity to the notation $Ax = b$ for matrix A and columns x and b. Our summation convention is simply the usual rule for matrix·multiplication combined with indicial notation.

Standard Form 1

Given: A row of coefficients of a linear form to be optimized

$$a_j^0 \qquad (j = 1, 2, \ldots, \ell)$$

A matrix of restraint coefficients

$$a_j^i \qquad (i = 1, 2, \ldots, m; \quad j = 1, 2, \ldots, \ell)$$

A column of constants

$$b^i \qquad (i = 1, 2, \ldots, m)$$

To find: A column of values for $x^j \geq 0$ $(j = 1, 2, \ldots, \ell)$ such that the variable x^0 is maximized subject to

$$(5.11) \qquad x^0 + a_j^0 x^j = 0$$

$$(5.12) \qquad a_j^i x^j \leq b^i .$$

Note that (5.11) is perfectly general since the sum $a_j^0 x^j$ may be minimized or maximized merely by changing the signs of the a_j^0. To convert (5.12) to equalities, we define the variables $x^{\ell+i} \geq 0$ $(i = 1, \ldots, m)$. Then (5.12) becomes

$$(5.12') \qquad a_j^i x^j + x^{\ell+i} = b^i .$$

If δ_h^i $(i=0,1, \ldots, m; \; h=0,1, \ldots, m)$ is the (m+1)-order <u>identity matrix</u> (essentially the Kronecker delta), then we can define

$$a_0^i = \delta_0^i \quad \text{and} \quad b^0 = 0$$

$$a_{\ell+h}^i = \delta_h^i \quad \text{for} \quad h = 1, \ldots, m$$

thus defining $j = 0$ and $j = \ell+1, \ldots, \ell+m = n$, and replace both (5.11) and (5.12') by

$$(5.13) \qquad a_j^i x^j = b^i \quad (i = 0, 1, \ldots, m; \; j = 0, 1, \ldots, n).$$

If the restraint equations can be put in the form (5.12'), then some of the b^i are permitted to be negative, if necessary. (See section X below.) However the number of $b^i < 0$ should be small to prevent an excessive number of iterations. If the restraints cannot be specified exactly in the form (5.12), then

the problem should be put in Standard Form 2.

Standard Form 2

Given: A row of coefficients of a linear form to be optimized

$$a_j^0 \qquad (j = 1, 2, \ldots, n)$$

A matrix of restraint coefficients

$$a_j^i \qquad (i = 2, 3, \ldots, m; \ j = 1, 2, \ldots, n)$$

A column of constants

$$b^i \geq 0 \qquad (i = 2, 3, \ldots, m)$$

To find: A column of values for $x^j \geq 0$ $(j = 1, \ldots, n)$ such that the variable x^0 is maximized subject to

$$(5.21) \qquad x^0 + a_j^0 x^j = 0$$

$$a_j^i x^j = b^i \quad (i = 2, 3, \ldots, m).$$

We now add to the system, artificially, the identity matrix (except δ_0^i which is always incorporated) and auxiliary variables x^{n+k} $(k = 1, \ldots, m)$ and construct an auxiliary form in which the variable x^{n+1} is to be maximized first, i.e. before maximizing x^0. This process is called Phase I.

The purpose of Phase I is to eliminate the artificially added columns from the system or at least to make sure that the corresponding variables become zero.* Hence we put a weight of unity on each one in the row of index 1, which will become the auxiliary form. Let $a_{n+1}^i = \delta_1^i$ and for $k = 2, 3, \ldots, m$ let $a_{n+k}^i = \delta_1^i + \delta_k^i$. The variables x^{n+k} are to be non-negative except for x^{n+1} which is defined by

$$(5.23) \qquad x^{n+1} + \sum_{k=2}^{m} x^{n+k} = 0 .$$

---

* At least one artificial column, usually the δ_1^i column, must remain in the solution (basis) at zero level even while maximizing the variable x^0.

the row 1 equation takes the form

$$a^1_j x^j + x^{n+1} = b^1 \qquad (j = 0, 1, \ldots, n). \tag{5.27}$$

Furthermore the columns a^i_0, a^i_{n+1}, a^i_{n+2}, ..., a^i_{n+m} now form δ^i_h .

The data assembly program computes a^1_j and b^1 automatically. If some, but not all, of the columns of δ^i_h occur in a^i_j ($j \leq n$), then these columns should be indicated as being in the initial basis. The corresponding rows will then be omitted in the sums (5.26). This is equivalent to adding artificial columns a^i_{n+k} only for those columns of δ^i_h which are missing.

The vectors a^i_0, a^i_{n+k} are never entered with the data. They are implicit in the code and, once eliminated, cease to exist. The basis headings, which are the j-indices of the columns a^i_j in position h = 0, 1, ..., m of the basis, are left zero for all artificial vectors since the position h identifies them sufficiently. Legitimate columns of δ^i_h, on the other hand, may go out of the basis and come back in out of position. Hence they must have names, as in Standard Form 1.

The column indices j = 1, ..., n are used above for expository purposes only. Any n distinct symbols can be used since they are only names, no particular ordering is implied. The same is not true, of course, of the row indices.

<u>Alternate Methods of Starting a Problem</u>

Experience has led to the incorporation of another device for obtaining initial solutions. Sometimes the formulator of the problem knows a feasible basis other than the identity. Provision is made for introducing, arbitrarily, any number of column vectors into the basis at the outset, with the machine making the decision as to which column of the basis each should occupy. If the formulator has misjudged and the specified columns produce a singular matrix, the machine prints out (and saves) the pertinent information and stops. If the resulting matrix is non-singular but the solution is infeas-

ible, then the composite algorithm automatically cuts in and works toward feasibility in succeeding transformations. (See section X below.)

One other provision of a less absolute nature has proved useful. Occasionally, a model is a re-work of an older problem so that something is known about its behavior. At other times the formulator has certain insights which he would like to exploit without committing himself to absolute statements regarding feasibility or singularity. It is possible to arrange the columns of a_j^i in order of decreasing likelihood of use, that is, with the most likely candidates for entry into a feasible or optimal basis occuring first. These may be separated from the others by a special mark (called a "curtain") which is equivalent to the following instruction: "If any candidate for entry into the basis is available ahead of this mark, use it; otherwise proceed to the others." Several marks may be used. They have often reduced the number of iterations required but when used injudiciously, i.e. on a mere hunch, may have the opposite effect.

There are, of course, other devices which require no special provisions except perhaps for loading. For example, certain activities (columns) may be withheld from the machine until optimality is obtained with the others. This is another advantage of the revised method and additional data may be added or subjected to the optimality test at any time, simply by re-loading.

VI - THE BASIS AND ITS INVERSE

Any non-singular square matrix formed from m+1 of the columns of a_j^i will be denoted by

$$\mu_h^i \quad (i = 0, 1, \ldots, m; \;\; h = 0, 1, \ldots, m; \;\; \mu_0^i = \delta_0^i\,)$$

where it is <u>always</u> assumed that $\mu_0^i = a_0^i = \delta_0^i$. The matrix μ_h^i is called the <u>basis</u> and it changes from iteration to iteration. One always begins the com-

putations with the initial basis

$$\mu_h^{i(0)} = \delta_h^i .$$

This imples that a_j^i (perhaps augmented artificially as in Standard Form 2) contains δ_h^i. Since the columns of δ_h^i may occur in any order in a_j^i, we use second-order subscripts on the j's to denote these column names, i.e.

$$(6.01) \quad \mu_h^{i(0)} = a_{j_h}^i = \delta_h^i \quad \text{initially.}$$

After T cycles, some of the columns of $\mu_h^{i(T)}$ will differ from δ_h^i. The columns have been drawn from a_j^i but if we write

$$(6.02) \quad \mu_h^{i(T)} = a_{j_h}^i$$

the j_h in (6.02) are not the same j_h as in (6.01). Hence it is necessary to tag the j_h with T and keep the definition of $j_h^{(T)}$ up to date. These indices are called <u>basis headings</u> and are referred to frequently.

We will not be so much concerned with the basis $\mu_h^{i(T)}$ on iteration T as with its inverse which will be denoted by $\pi_h^{i(T)}$, that is

$$(6.1) \quad \pi_k^{i(T)} \mu_h^{k(T)} = \mu_k^{i(T)} \pi_h^{k(T)} = \delta_h^i .$$

The matrix $\pi_h^{i(T)}$ never exists explicitly but is carried as a product of elementary matrices which are stored in a condensed form consisting of at most one column plus an index and identification. These columns are themselves condensed with only non-zero elements recorded. They are called <u>transformation vectors</u> (sometimes elimination "equations", see (6.5)) and denoted by $\eta_r^{i(t)}$. Since the index r is itself a function of t, it should be written r(t). However this will not usually be done since it sometimes appears as a superscript in an array already indexed (t-1) and confusion would result.

As already indicated, Greek letters will be used for a basis and its inverse, and also for all (m+1)-order vectors expressed in terms of the basis. Likewise certain ratios and functions involved in decisions concerning the

basis will be denoted by Greek letters.

An explanation of the generation and use of the $\eta_r^{i(t)}$ vectors is in order. Since we started with $\mu_h^{i(0)} = \pi_h^{i(0)}$ for the first iteration, we will have $\pi_h^{i(T-1)}$ on hand to start iteration T. Suppose that column $a_{s(T)}^i$ has been chosen from a_j^i to replace column $h = r(T)$ in $\mu_h^{i(T-1)}$, producing a new basis $\mu_h^{i(T)}$. Throughout the following discussion, the specific index r is to be understood as r(T). Superscripts in parentheses will always refer to the entire array to which the element belongs and are not to be understood as modifying the open superscript.

Let $\alpha_{s(T)}^i$ satisfy the equation

$$(6.2) \qquad \mu_h^{i(T-1)} \alpha_{s(T)}^h = a_{s(T)}^i .$$

Clearly $\alpha_{s(T)}^i$ can be obtained by

$$(6.3) \qquad \pi_h^{i(T-1)} a_{s(T)}^h = \alpha_{s(T)}^i .$$

Now

$$(6.4) \qquad \mu_k^{i(T)} \delta_h^k = \mu_h^{i(T-1)} \quad \text{for } h \neq r(T)$$

since only column $h = r(T)$ changes. However, from (6.2) we can solve for the exceptional column.

$$(6.5) \qquad \mu_r^{i(T-1)} = \frac{1}{\alpha_{s(T)}^r} a_{s(T)}^i + \mu_h^{i(T-1)} \eta^h \qquad (h \neq r)$$

where

$$(6.6) \qquad \eta^h = \frac{-\alpha_{s(T)}^h}{\alpha_{s(T)}^r} \qquad (\alpha_{s(T)}^r \neq 0 \text{ by choice}).$$

Now defining

$$(6.7) \qquad \eta_r^{r(T)} = \frac{1}{\alpha_{s(T)}^r}, \quad \eta_r^{i(T)} = \eta^i \quad (i \neq r)$$
$$\eta_h^{i(T)} = \delta_h^i \qquad (h \neq r)$$

we can replace (6.4) and (6.5) by

$$(6.8) \qquad \mu_k^{i(T)} \eta_h^{k(T)} = \mu_h^{i(T-1)}$$

since $a_{s(T)}^i$ becomes column $\mu_r^{i(T)}$. Clearly only the column $\eta_r^{i(T)}$ and the index $r = r(T)$ need be recorded. Now multiplying both members of (6.8) on the left

by $\pi_i^{j(T)}$ and on the right by $\pi_k^{h(T-1)}$, we obtain

$$(6.9) \qquad \eta_h^{j(T)} \pi_k^{h(T-1)} = \pi_k^{j(T)} .$$

Equation (6.9) is the heart of the product form of inverse. Applying it for $t = 1, 2, \ldots, T$ and using h_t to indicate dummy indices, we obtain

$$(6.10) \qquad \pi_h^{i(T)} = \eta_{h_{T-1}}^{i(T)} \eta_{h_{T-2}}^{h_{T-1}(T-1)} \cdots \eta_{h_{t-1}}^{h_t(t)} \cdots \eta_h^{h_1(1)} .$$

Hence an equation like (6.3) implies the recursive generation of its right member by using the form of $\pi_h^{i(T)}$ given by (6.10). This is easily done as follows, using (6.3) for an example.

$$(6.11) \qquad \begin{array}{ll} \text{Let} & a_{s(T)}^{i} = \bar{\alpha}_{s(1)}^{i} \\ \text{form} & \eta_h^{i(1)} \bar{\alpha}_{s(1)}^{h} = \bar{\alpha}_{s(2)}^{i} \\ & \vdots \\ & \eta_h^{i(t-1)} \bar{\alpha}_{s(t-1)}^{h} = \bar{\alpha}_{s(t)}^{i} \\ & \vdots \\ & \eta_h^{i(T-1)} \bar{\alpha}_{s(T-1)}^{h} = \bar{\alpha}_{s(T)}^{i} = \alpha_{s(T)}^{i} \end{array}$$

It is easy to see that

$$(6.12) \qquad \begin{array}{ll} \bar{\alpha}_{s(t)}^{i} = \bar{\alpha}_{s(t-1)}^{i} + \bar{\alpha}_{s(t-1)}^{r} \eta_r^{i(t-1)} & (i \neq r = r(t-1)) \\ \bar{\alpha}_{s(t)}^{r} = \qquad \bar{\alpha}_{s(t-1)}^{r} \eta_r^{r(t-1)} & (r = r(t-1)) \end{array}$$

An equally simple rule suffices for multiplying $\pi_h^{i(T)}$ by a row vector on the left. Suppose it is necessary to compute

$$(6.13) \qquad f_i \pi_h^{i(T-1)} = \pi_h^{(T)} .$$

We use the transformation vectors in reverse order to that of (6.11) :

$$(6.14) \qquad \begin{array}{ll} \text{Let} & f_i = \bar{\pi}_h^{(1)} \\ \text{form} & \bar{\pi}_h^{(1)} \eta_i^{h(T-1)} = \bar{\pi}_i^{(2)} \\ & \vdots \end{array}$$

$$\vdots$$

(6.14 cont.)

$$\bar{\pi}_h^{(t)} \eta_i^{h(T-t)} = \bar{\pi}_i^{(t+1)}$$

$$\vdots$$

$$\bar{\pi}_h^{(T-1)} \eta_i^{h(1)} = \bar{\pi}_i^{(T)} = \pi_i^{(T)}$$

Again it is easy to see that

(6.15)

$$\bar{\pi}_i^{(t+1)} = \bar{\pi}_i^{(t)} \qquad (i \neq r(T-t))$$

$$\bar{\pi}_r^{(t+1)} = \bar{\pi}_i^{(t)} \eta_r^{i(T-t)} \qquad (r = r(T-t))$$

VII - THE PRICING OPERATION (Choosing index s)

The row vector $\pi_i^{(T)}$ in (6.13) is called the pricing vector. In the simplest problem (Phase II, maximize x^0, no infeasibilities), $f_i = \delta_i^0$. However, in a typical Phase I, $f_i = \delta_i^1$, and in general $f_i = 1$ for several i.

The pricing vector is applied to a_j^i,

(7.1) $$\pi_i^{(T)} a_j^i = d_j^{(T)} .$$

(The d_j are what are called $(\pm)$ $(c_j - z_j)$ in the original simplex method.) To choose $a_{s(T)}^i$ to bring into the basis, take

(7.2) $$d_s^{(T)} = \min d_j^{(T)} < 0 .$$

(If the minimum is not unique, the first such index s is retained.) If all $d_j^{(T)} \geq 0$, the phase is complete. In a Phase I, this is Terminal 1; in a Phase II it is called Terminal 2 and is the point at which one usually expects to arrive and quit, i.e. the optimal solution is attained.

VIII - CHOOSING THE BASIS COLUMN TO BE REPLACED (Index r)

Let the current basis be $\mu_h^{i(T-1)}$, the current optimizing form be row p and the current solution vector be $\beta^{i(T-1)}$, i.e.

(8.1) $$\beta^{i(T-1)} = \pi_h^{i(T-1)} b^h .$$

We will assume the solution is feasible, that is

(8.2) $$\beta^{i(T-1)} \geq 0 \quad \text{for } i \geq q$$

where row q is the first actual restraint equation. Normally $q = p+1$.

The usual criterion for choosing $r(T)$ is to choose $\theta_{r(T)}$ as follows. Let A be the set of indices $i \geq q$ for which $\alpha^i_{s(T)} > 0$. Then

$$(8.3) \qquad \theta_{r(T)} = \min_{i \in A} \left\{ \frac{\beta^{i(T-1)}}{\alpha^i_{s(T)}} \right\} \geq 0 \; .$$

The problem of degeneracy, i.e. multiple values of $i \in A$ for which $\beta^{i(T-1)} = 0$, is disregarded except for the following rule which has proved effective for reducing round-off error. (It incidentally breaks the tie in Hoffman's examples of "cycling.") If $\theta_{r(T)} = 0$ and $r(T)$ is ambiguous, choose $r(T)$ so that $\alpha^{r(T)}_{s(T)}$ is the largest possible (positive) value. In case of further ambiguity, take the smallest such index.

We will modify (8.3) slightly. Let R be the set of indices $i \geq q$ for which $\beta^{i(T-1)}$ and $\alpha^i_{s(T)}$ have the same sign with $\alpha^i_{s(T)} \neq 0$. Then let

$$(8.4) \qquad \theta_{r(T)} = \min_{i \in R} \left\{ \frac{\beta^{i(T-1)}}{\alpha^i_{s(T)}} \right\} \geq 0 \; .$$

Note that (8.4) gives the same result as (8.3) as long as (8.2) holds. Degeneracy is resolved in the same way, that is, if $\theta_{r(T)} = 0$, take $\alpha^{r(T)}_{s(T)} > 0$ and max.

If no $\theta_{r(T)}$ can be chosen, that is, all ratios are non-positive and zero ratios have negative denominators, then β^p has no finite maximum. A class of solutions exist as follows:

$$(8.5) \qquad \mu_h^{i(T-1)} \left(\beta^{h(T-1)} - \theta\, \alpha^h_{s(T)}\right) + \theta\, a^i_{s(T)} = b^i$$

with the value

$$(8.6) \qquad \beta^{p(T-1)} - \theta\, \alpha^p_{s(T)} \longrightarrow +\infty \text{ as } \theta \longrightarrow +\infty \; .$$

This is Terminal 3. It cannot happen in a Phase I since clearly zero is an upper bound for the variable x^{n+1}.

Clearly, $x^{n+1} \leq 0$ and, if $x^{n+1} = 0$, then all $x^{n+k} = 0$. When (and if) this condition is attained, the artificial variables are maintained at zero, including x^{n+1}, by considering (5.23) as a restraint while maximizing x^0 (Phase II.) If x^{n+1} cannot be driven to zero, there is no solution to the given problem. This condition is called Terminal 1.

The problem at this point can be displayed in the following augmented form where the x^j are shown above the matrix of detached coefficients.

$$\begin{matrix} x^0 & x^1 & \dots & x^n & x^{n+1} & x^{n+2} & x^{n+3} & \dots & x^{n+m} \end{matrix}$$

$$(5.24) \qquad \begin{bmatrix} 1 & a_1^0 & \dots & a_n^0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 1 & 1 & 1 & \dots & 1 \\ 0 & a_1^2 & \dots & a_n^2 & 0 & 1 & 0 & \dots & 0 \\ 0 & a_1^3 & \dots & a_n^3 & 0 & 0 & 1 & \dots & 0 \\ & - & - & - & & & \cdot & & \\ & & & & & & & \cdot & \\ & & & & & & & & \cdot \\ 0 & a_1^m & \dots & a_n^m & 0 & 0 & 0 & \dots & 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ b^2 \\ b^3 \\ \cdot \\ \cdot \\ \cdot \\ b^m \end{bmatrix}$$

Since the identity matrix still does not appear in (5.24), we must make a simple preliminary transformation. First note, however, that setting

$$(5.25) \qquad \begin{aligned} & x^j = 0 \quad \text{for} \quad j = 0, 1, \dots, n \\ & x^{n+1} = -\sum_{i=2}^{m} b^i \quad \text{and} \quad x^{n+i} = b^i \quad \text{for} \quad i = 2, 3, \dots, m \end{aligned}$$

provides an initial solution in which all variables are non-negative except x^{n+1}. This solution will remain valid if we subtract all equations of index 2 through m from the auxiliary optimizing form, row 1. Then defining anew

$$(5.26) \qquad \begin{aligned} & a_j^1 = -\sum_{i=2}^{m} a_j^i \quad \text{for} \quad j = 1, \dots, n; \qquad b^1 = -\sum_{i=2}^{m} b^i \\ & a_{j+k}^1 = 0 \quad \text{for} \quad k = 2, \dots, m \end{aligned}$$

IX - SUMMARY OF CYCLIC OPERATIONS (One Iteration)

(9.0) Test for end of a Phase I or for arbitrary halt (external switch.)

(9.1) Determine values of f_i (discussed further below.)

(9.2) Form $\pi_i^{(T)} = f_h \pi_i^{h(T-1)}$ by (6.14).

(9.3) Compute $d_j^{(T)} = \pi_i^{(T)} a_j^i$ and choose $d_{s(T)}^{(T)} = \min d_j^{(T)} < 0$ or terminate if all $d_j^{(T)} \geq 0$.

(9.4) Compute $\alpha_{s(T)}^i = \pi_h^{i(T-1)} a_{s(T)}^h$ by (6.11).

(9.5) Choose $\Theta_{r(T)}$ by (8.4) or terminate.

(9.6) Compute $\eta_r^{i(T)}$ by (6.6,7), transform $\beta^{i(T-1)}$ to $\beta^{i(T)}$ by one step as in (6.11) and record s(T) for $j_{r(T)}^{(T)}$.

(9.7) (Optional) Print results of iteration (See Part B).

(9.8) Condense and store $\eta_r^{i(T)}$ and r(T).

(9.9) (Optional) Check solution and print (See Part B).

X - THE COMPOSITE ALGORITHM (Forming f_i)

Suppose that a basic solution has been obtained which is infeasible, that row p is the current optimizing form and that q is the smallest row index of the actual restraint equations.

(10.1) $\mu_h^{i(T-1)} \beta^{h(T-1)} = b^i$, $h \in F$ if $h \geq q$ and $\beta^{h(T-1)} < 0$.

Suppose a vector $a_{s(T)}^i$ has somehow been chosen to bring into the basis and that (8.4) is used to determine r(T). After the change of basis, the new values of β^i are

(10.2) $\beta^{r(T)} = \Theta_{r(T)} \geq 0$ (the value of $x^{s(T)}$)

(10.3) $\beta^{i(T)} = \beta^{i(T-1)} - \Theta_{r(T)} \alpha_{s(T)}^i$ $(i \neq r(T))$

Now for $i \geq q$ and $i \notin F$, (10.3) gives $\beta^{i(T)} \geq 0$. However, for $i \in F$, there are two cases.

(10.4) If $i \in F$ and $\alpha^i_{s(T)} < 0$, then $\beta^{i(T)} \geq \beta^{i(T-1)}$

(10.5) If $i \in F$ and $\alpha^i_{s(T)} \geq 0$, then $\beta^{i(T)} \leq \beta^{i(T-1)}$

Hence by (10.2) and (10.3), no infeasibilities are created. By (10.2) and (10.4), some infeasibilities are improved or removed altogether. However, by (10.5) some may be made worse. To overcome this difficulty, consider the function $\lambda = \sum_{i \in F} \beta^i \leq 0$ which is a measure of the infeasibility of a solution. We wish to maximize it to zero and hence we may replace the maximization of β^p by the maximization of

(10.6) $\sigma = \beta^p + \lambda \leq \beta^p$

provided σ is monotonically non-decreasing. When σ reaches its maximum, if $\lambda = 0$ then β^p is maximum. If $\lambda < 0$, then β^p is too great and λ must be increased without regard to decreases in β^p.

Let $f_i = 1$ if $i \in F$ or $i = p$, $f_i = 0$ otherwise. Then, as can be seen from (9.2,3,4)

(10.7) $d_s^{(T)} = f_i \; \alpha^i_{s(T)} < 0$.

Consequently, after change of basis, the new value of σ is *

(10.8) $\sigma^{(T)} = \sigma^{(T-1)} - \theta_{r(T)} \; d_s^{(T)} \geq \sigma^{(T-1)}$.

If the set F is void, then (10.8) is the same as

(10.9) $\beta^{p(T)} = \beta^{p(T-1)} - \theta_{r(T)} \; d_s^{(T)} \geq \beta^{p(T-1)}$

which is the usual formula for the change in the maximand.

Now, however, if all $d_j^{(T)} \geq 0$, we cannot claim Terminal 2 without checking to see whether F is void. If F is not void and all $d_j^{(T)} \geq 0$, it may be because (10.9) dominates (10.8).[θ] In this case, we may scale down f_p , perhaps

---

* Note that if $\beta^{i(T-1)} < 0$, then $\beta^{i(T)} \leq 0$ for $i \in F$ and $i \neq r(T)$.

θ That is, $\pi_i^{p(T-1)} \; a^i_j \geq \left| f_h \; \pi_i^{h(T-1)} \; a^i_j \right|$ for $h \in F$.

to zero, until such time as $\lambda = 0$. If all $d_j^{(T)} \geq 0$ with $f_p = 0$, then no feasible solution exists since $\lambda < 0$ is at a maximum.

Even though $\lambda = 0$, it is sometimes desirable to set $f_p < 1$. Of course, tolerances must be built into the code in testing $d_j \geq 0$ since, if it is sufficiently small in magnitude, it ought to be considered zero. Varying f_p has the effect of varying this built-in tolerance. Provision has also been made for setting $f_p < 0$ which causes β^p to be minimized instead of maximized. This is often convenient when experimenting with a model. The value of f_p is entered on a binary card subject to a switch.

If $\sigma < \beta^p$ is at a maximum and $f_p \neq 0$, the machine will set $f_p = 0$ and stop so that other values may be loaded if desired. If $\lambda = 0$ and $f_p = 0$, the machine will set $f_p = 1$ and stop.

The variable β^p is checked for monotonic behavior each iteration, according as $f_p > 0$ or $f_p < 0$. This test is suspended if $\lambda < 0$ or when making arbitrary transformations when the behavior of β^p is unpredictable.

XI - PARAMETRIC PROGRAMMING (PLP)

Provision can be made for parametrizing the right hand side as a linear function of $\Theta \geq 0$, i.e.

$$(11.1) \qquad a_j^i x^j = b^i + \Theta c^i$$

where, if a Phase I was used, c^1 must be formed in the same manner as b^1 in (5.26).

To do this, first find an optimal solution for $\Theta = 0$, say

$$(11.2) \qquad \mu_h^{i(T-1)} \beta^{h(T-1)} = b^i = b^{i(T-1)} .$$

Let

$$(11.3) \qquad -\pi_h^{i(T-1)} c^h = \gamma^i .$$

Now using γ^i in place of $\alpha_{s(T)}^i$ in (8.4), a value $\Theta_{r(T)} = \Delta\Theta$ can be found such that

$$(11.4) \qquad \mu_h^{i(T-1)} \left(\beta^{h(T-1)} - \theta_{r(T)} \gamma^h \right) = b^i + \theta_{r(T)} c^i$$

with

$$(11.5) \qquad \beta^{i(T-1)} - \theta_{r(T)} \gamma^i \geq 0 \quad \text{for } i \geq q$$

and

$$\beta^{r(T-1)} - \theta_{r(T)} \gamma^r = 0 \quad (r = r(T)).$$

The parameter θ cannot be increased by more than $\theta_{r(T)}$ with the basis $\mu_h^{i(T-1)}$ without violating (11.5) but (11.4) is an optimal feasible solution to (11.1) for this value of θ. Let

$$(11.6) \qquad \begin{aligned} b^{i(T)} &= b^{i(T-1)} + \theta_{r(T)} c^i \\ \beta^{i(T)} &= \beta^{i(T-1)} - \theta_{r(T)} \gamma^i . \end{aligned}$$

To increase θ further, $\mu_{r(T)}^{i(T-1)}$ must be removed from the basis and replaced with some $a_{s(T)}^i$ to form a new basis $\mu_h^{i(T)}$ so that

$$(11.7) \qquad \mu_h^{i(T)} \beta^{h(T)} = b^{i(T)}$$

is also an optimal feasible solution to (11.1) for the same θ. The whole process can then be repeated.

The index $s(T)$ is determined by the rule used in the dual simplex algorithm. Let D be the set of indices j for which $\pi_h^{r(T-1)} a_j^h < 0$ $(r = r(T))$. Then choose $\varphi_{s(T)}$ by

$$(11.8) \qquad \varphi_{s(T)} = \min_{j \in D} \left\{ \frac{\pi_h^{p(T-1)} a_j^h}{- \pi_h^{r(T-1)} a_j^h} \right\} \geq 0 \qquad (r = r(T)) .$$

Note that

$$\varphi_{s(T)} = \frac{d_s^{(T)}}{-\alpha_{s(T)}^r}$$

and (11.8) is the analogue in the dual problem of (8.3).

If the set D is void, then θ is at a maximum. If all $\gamma^i < 0$ in (11.3), then θ can be increased without bound. These are the only two automatic

terminations in PLP. The following theorem is of interest.

Theorem: If the choice of $r(T)$ for (11.4) is unique and if D is not void, then there exists a finite range of θ, $\theta_{r(T)} \leq \theta \leq \theta_{r(T)}+\epsilon$, for which the solution obtained by replacing $\mu^{i(T-1)}_{r(T)}$ by $a^{i}_{s(T)}$, where $s(T)$ is determined by (11.8), is both feasible and optimal.

A proof can be found in Reference 4.

A separate control code for the computer is used in doing PLP. It always starts from a prior optimal, feasible solution. Due to (11.8), the iterations are longer than the regular code.

XII - MULTIPLE PHASES

It is somewhat difficult to parametrize the optimizing form since the analogue of γ^{i} would be a row vector of n+1 elements. As an alternative, provision is made for multiple optimizing forms which can be made to differ by finite amounts in any desired way. Of course, it is not contemplated that two such forms will be drastically different since that is equivalent to two different problems. In such a case, it would be better to start the second one from the beginning or from the end of Phase I.

It is also possible to split Phase I into multiple phases. This will not be discussed further since its application is limited and it generalizes easily from the discussions given.

It will be easier to describe the use of multiple phases if a specific example is used. Let it be required to optimize three forms and to start the problem with a Phase I. Then the auxiliary form must be

$$(12.1) \qquad a^{3}_{j}\, x^{j} + x^{n+1} = 0 \qquad (j = 3, 4, \ldots, n).$$

The form to be optimized first (after Phase I) must be

$$(12.2) \qquad x^{2} + a^{2}_{j}\, x^{j} = 0 \quad (\text{maximize } x^{2}).$$

Similarly, the other two forms to be optimized in turn must be

$$(12.3) \qquad x^1 \quad + a_j^1 x^j = 0 \quad \text{(maximize } x^1\text{)}$$

$$(12.4) \qquad x^0 \quad + a_j^0 x^j = 0 \quad \text{(maximize } x^0\text{)} .$$

The initial restraint equations will be

$$(12.5) \qquad a_j^i x^j + x^{n+i} = b^i \qquad (i = 4, 5, \ldots, m; \; j = 3, 4, \ldots, n).$$

Thus the initial value of q is specified as q = 4 (total number of phases) giving p = q-1 = 3 (number of " Phase II's"). The other two parameters required are z = 1 (number of " Phase I's") and $\xi = 3$ (index of sum row.)

At the end of Phase I, p, q and z will all be reduced by 1 giving p =2, q = 3, z = 0 so that x^2 will now be maximized disregarding x^1 and x^0, (12.1) will now be considered a restraint equation the same as (12.5), and the phase will be recognized as a Phase II.*

When x^2 reaches a maximum, p will be reduced by 1 to p = 1 but q will remain at 3 so that x^1 will be maximized disregarding x^2 and x^0. Similarly, when x^1 reaches a maximum, p will be reduced to p = 0 with q still remaining at 3 so that x^0 will be maximized disregarding x^2 and x^1. In other words, p is reduced each phase but q is reduced each phase only as long as z can also be reduced. All three must remain non-negative, obviously.

---

* A Phase I is terminated when the variable being maximized reaches zero. A Phase II terminates when all $d_j \geq 0$. These criteria are not always equivalent even in Phase I. If b^i is representable with fewer than m of the columns a_j^i (i, j > 0), then several artificial columns may remain in the basis at zero level at the end of Phase I. In this case, the d_j corresponding to the Phase I pricing vector will not all be non-negative. If Phase II starts with artificial columns in the basis (other than a_0^i and a_{n+1}^i), then a_{n+1}^i may be eliminated in Phase II but one a_{n+k}^i will always remain.

PART B - ORGANIZATION OF THE CODES

We will describe the organization of the present programs for the IBM 704 which are a culmination of the experience previously discussed. However, the discussion will be tied to the machine only to the extent of allowing specific statements to be made. Alternate techniques for other machines should be apparent to anyone with sufficient familiarity with computers to have a real interest. The equipment assumed available is the standard 704 with 4,096 words of core storage, four drums of 2,048 words each, card reader, card punch, printer and five magnetic tape units. The data assembly program will be discussed only briefly.

The following parameters must be specified:

- m, the total number of restraint equations including all but the last optimizing form (row 0.) The maximum m is 255.
- q, the total number of phases.
- z, the number of Phase I's. (Usually 0 or 1.)
- ξ, the index of the auxiliary form (sum row) for Phase I, blank indicating none.
- τ, the number of arbitrary transformations.

The data assembly program reads an identification card followed by a card with these parameters and then by cards containing, in order:

Initial basis headings, if any, in which any vectors to be arbitrarily introduced are also specified by making $j_h^{(0)}$ negative.

The right hand side b^i.

Auxiliary right hand sides c^i, if any.

The elements a_j^i by columns.

Only non-zero elements are entered, with their proper indices. The program punches in binary cards:

The identification card (in binary coded decimal-alphabetic)

A self loading origins card needed by the loader for the main routines (2(m+1), origins for H-, V-, W-, T-regions. See below.)

That part of K-region which is peculiar to the problem. (See below.)

The b^i in double precision, floating point.

The a^i_j matrix is packed and written on tape in single precision, fixed point, indexed form. It may also be punched on cards if desired.

There are, of course, provisions for various circumstances and certain built-in checks, as for example, computing and inserting the sum row in a^i_j and checking that no row index is greater than m. The main point is that considerable thought must go into the planning of this program to make the operation of the main codes as efficient as possible. We turn to them now.

Conceptually, the high-speed store (HSS) is divided into two main sections: (i) the programs, constants, parameters and temporary storage, (ii) the data, both original and transformed.

Similarly, auxiliary storage is divided into two parts, one part as permanent storage for (i), the other as permanent storage for (ii). Thus the entire HSS is in a sense "temporary storage". In practice, a considerable part of section (i) of HSS remains intact throughout a run but can be restored from auxiliary storage (drums) if necessary. The advent of the extremely reliable core-storage, however, makes the need of restoring unlikely. When HSS reaches the sizes now contemplated (or as much as 12,288 words), auxiliary storage for the code and for certain data will be unnecessary. The main use for auxiliary storage is for the a^i_j matrix and the $\eta^{i(t)}_r$ vectors. Magnetic tapes are used for both. On the IBM 704, three tapes are actually

used for the $\eta_r^{i(t)}$ vectors but on some tape systems this would be unnecessary.

The data section of HSS is divided into four regions:

the H-region for the basis headings: $m+1$ words

the V-region for the values β^i of the current solution, maintained in double-precision, floating point: $2(m+1)$ words

the W-region for work space in generating $\alpha^i_{s(T)}$ and $\pi_i^{(T)}$ and other purposes: $2(m+1)$ words.

the T-region for temporarily holding the a^i_j matrix or $\eta_r^{i(t)}$ vectors or as much as possible of either: the remainder of section (ii) which should be at least $4(m+1)$ words.

The size of these regions is a function of the number of restraints. Their origins are computed by the data assembly program. In PLP, V-region is used as a second W-region in pricing. A duplicate of H- and V-regions, as well as the original b^i, is kept in auxiliary storage. This allows re-starting after an error and checking a solution by computing and printing

$$\epsilon^i = \mu_h^{i(T)} \beta^{h(T)} - b^i.$$

For PLP, it is also necessary to keep c^i and γ^i in auxiliary storage.

The program section of HSS is divided into seven regions:

(a) Temporary storage called COMMON.

(b) The main control region, called the MCR.

(c) A sub-routine for doing double precision, floating point addition, called DPFADD.

(d) A similar sub-routine for multiplication, called DPFMUL.

(e) A routine called the distributor (DISTRB), explained below.

(f) Space for the largest sub-routine. The first location is called SRORIG,

(g) Universal constants, parameters, origins etc., called K-region. The need for COMMON, DPFADD, DPFMUL and K-region is obvious. Their locations are permanently fixed and may be referred to from any program in the system. All sub-routines are closed. The K-region contains certain cells whose contents are fixed only for part of an iteration or other sub-sequences of the problem. There are conventions on when and by what routine these are to be changed. COMMON is always available to any routine.

The function of the MCR is to make the major decisions and call for the proper sequence of operations. Most of the actual work is delegated to sub-routines. Besides the MCR for the main algorithm, there is one for PLP and one for re-inverting a basis. Other auxiliary programs are easily created by coding a new MCR.

The MCR calls for a major operation by linking to DISTRB with a pseudo-operation. DISTRB calls in the proper sub-routine from auxiliary storage and loads it into HSS starting at SRORIG. Control is then transferred to the sub-routine which returns control to the MCR after completing its function. If some other arrangement for linking to sub-routines is desired (as for example, when HSS is very large) it is only necessary to change DISTRB to arrange for the proper routine to take over in the proper way.

There are fourteen standard sub-routines. Others may be added for special purposes if desired, up to the limit of auxiliary storage to hold them. More importantly, if an improved or a special version of one is developed, it can replace the old one merely by exchanging the proper binary cards. These sub-routines, together with DPFADD, DPFMUL, K-region and a special loading routine, exist in binary cards which constitute a basic deck. The origins card produced by the data assembly is put ahead of, and the MCR

behind this basic deck.

As soon as the special loading routine is in HSS, it takes control and loads the remainder of the basic deck and the MCR, storing them in auxiliary storage (drums). Since many of the sub-routines require addresses to be set which depend upon m, a loading interlude is performed to do this initialization. This is accomplished by an initializing routine which goes with a sub-routine, the two being loaded simultaneously into HSS. Control is then sent to the initializing routine which does its job once and for all and returns control to the loader*. The loader stores the initialized sub-routine in auxiliary storage and builds a catalogue of locations into DISTRB. The MCR takes over control when loading is complete.

The fourteen sub-routines (called "codes") are as follows. Some are used for multiple purposes which are controlled by internal switches.

Code 1. Load the binary cards produced by the data assembly and store in appropriate places. If the a_j^i cards are included, they are transferred to tape. Subject to a switch, a value may be loaded from a special card for f_p. By means of an internal switch and the contents of K-region which it loads, this load routine is able to distinguish various cases, initial start, re-start, start of PLP, re-start of PLP, so that the proper information can be stored in auxiliary and the tapes positioned properly.

Code 2. Store the current solution in auxiliary (drums). The "current solution" is defined as K-, H-, and V-regions.

Code 3. (Normal) Form the row f_i in W-region, and record whether any variables are infeasible.

(During PLP) Form δ_i^p in V-region and δ_i^r in W-region.

---

*In so far as the linear programming codes under discussion are concerned, initialization is an inovation in the 704 system due to Hal Judd of IBM.

Code 4. (Normal) Compute $\pi_h^{(T)} = f_i \; \pi_h^{i(T-1)}$ in W-region.

(During PLP) Compute $\pi_i^{p(T-1)}$ in V-region and $\pi_i^{r(T-1)}$ in W-region.

Code 5. (Normal) Price the matrix a_j^i and choose $d_s^{(T)}$.

(During PLP) Choose $\varphi_{s(T)}$ as described in Part A, Sect. XI.

Subject to an external switch, recognize "Curtain" marks in a_j^i.

Subject to instructions from the MCR, makd the following check:

(Normal) If $d_j^{(T)} \neq 0$, (during PLP) if $\pi_i^{r(T-1)} \; a_j^i \neq 0$ for $j \neq j_{r(T)}^{(T-1)}$

then test for j occurring in $j_h^{(T-1)}$. If it does, an error has occurred. This check is very valuable for detecting errors before much more computing is done. It does consume some amount of time, however, which increases with n.

Code 6. Load $a_{s(T)}^i$ from a_j^i matrix into specified region as double-precision, floating point vector.

Code 7. Multiply $\pi_h^{i(T-1)} \; a_{s(T)}^h$ (or any vector in specified region).

Code 8. (Normal) Choose the index r(T) as discussed in Part A, Sect. VIII. For arbitrary transformations or inversion, choose r(T) by:

$$\left| \alpha_{s(T)}^r \right| = \max_{i \in A} \left| \alpha_{s(T)}^i \right|$$

where A is the set of indices i for which $j_i^{(T-1)} = 0$ (artificial).

Code 9. (Normal) Compute $\eta_r^{i(T)}$ from $\alpha_{s(T)}^i$ and transform $\beta^{i(T-1)}$ to $\beta^{i(T)}$.

(PLP, first entry) Compute $\beta^{i(T)}$ and $\theta_{r(T)}$ from $\beta^{i(T-1)}$, r(T) and $\gamma^{i(T-1)}$. (Part A, Sect. XI). Then compute $b^{i(T)} = b^{i(T-1)} + \theta_{r(T)} c^i$.

(PLP, second entry) Compute $\eta_r^{i(T)}$ from $\alpha_{s(T)}^i$ and transform $\gamma^{i(T-1)}$ to $\gamma^{i(T)}$. (All these operations are very similar, even more than appears at first glance. The variations are merely switches.)

Code 10. Delete zeros from $\eta_r^{i(T)}$ and index non-zero elements. Store condensed vector in auxiliary storage (drums; cf. Code 13).

Code 11. Multiply out $\mu_h^{i(T)} \beta^{h(T)} - b^i = \epsilon^i$ taking $\mu_h^{i(T)}$ from original a_j^i matrix by referring to $j_h^{(T)}$. ϵ^i is left in T-region and b^i in W-region for printing.

Code 12. Print program. This program is quite elaborate and longer than the other codes. Special provisions for it need not be discussed here except to say that appropriate captions and identification of columns are printed for each type of print-out, of which there are 14. The print output can also be put on the fifth tape, if desired, for later printing. In all print-outs, there are 5 columns of which the first three are: $j_i^{(T)}$, $\beta^{i(T)}$ and $i = 0, \ldots, m$. The last two are the contents of W- and T-regions.

Code 13. This performs an "end-of-stage" in which some of the $\eta_r^{i(t)}$ are transferred from drum to tape. See below.

Code 14.-18. (Undefined).

Code 19. Automatic restart program for recovering after an error by returning to the beginning of the iteration. Clearly, this is highly dependent on the particular machine. The important points to note are that all programs, the original data, the current solution at beginning of the iteration, and the $\eta_r^{i(T)}$ vectors must

be intact somewhere in the machine. Also this code must have been previously instructed where to return control to and be able to restore all equipment (e.g. tapes) to the proper positions. It must, of course, be activated by some external means.

Besides the physical organization above, there are also dynamic groupings, several of which have already been discussed: the iteration, the phase, maximization of $\lambda<0$, arbitrary transformations, etc. Another grouping is called a stage and always consists of an integral number of iterations. It has nothing to do with phases or other mathematical aspects but is simply an operational device. It was devised for the IBM 701 and has been carried over to the 704, for slightly different reasons. Some analagous arrangement is probably necessary on any machine with non-homogeneous storage media.

Most of the advantage of condensing the $\eta_r^{i(t)}$ vectors would be lost if a separate access to auxiliary storage had to be made for each one. It is desirable that as big a chunk of these transformation vectors as possible be recalled at one time. The limiting factor is the size of T-region. Hence, as the $\eta_r^{i(t)}$ are generated, they are stored on a drum until one more would exceed the capacity of T-region. At this point an end-of-stage procedure is performed. Before describing this, it is necessary to explain the use of three tapes for the $\eta_r^{i(t)}$ vectors.

The tapes on the 704 can be back-spaced and additional records can be added to an existing file. However, they cannot be read in a backward direction and the back-space is somewhat slow. Hence the records are stored in reverse order on a second tape for use in computing $\pi_i^{(T)}$. The third tape is used alternately with the second from stage to stage. Though this costs some copying time, it does provide a spare (and checked) copy of the tape at

last stage but one which is very desirable. It is dangerous for one to burn all his bridges behind him, especially with magnetic tapes.

The end-of-stage procedure is as follows:

(1) Compute ϵ^i.

(2) Print $j_i^{(T)}$, $\beta^{i(T)}$, i, b^i, ϵ^i.

(3) Transfer the $\eta_r^{i(t)}$ on drum to an additional record on the forward tape.

(4) Transfer the $\eta_r^{i(t)}$ on drum to the first record on the free backward tape.

(5) Copy old backward tape to remainder of new backward tape.

(6) Punch out binary cards containing K-region, $j_i^{(T)}$ and $\beta^{i(T)}$. During PLP, also punch $b^{i(T)}$ and $\gamma^{i(T)}$.

(7) Adjust the necessary bookkeeping parameters.

An end-of-stage may be forced by an external switch. It also occurs at the end of Phase I and terminations. Steps (1) and (2) above may be forced without the others at the end of a cycle by an external switch. This would be desirable on machines which do not require the above tape manipulations.

To restart from the end of a stage, it is only necessary to use the punch-outs to replace the corresponding original data cards and put the tapes back on the same units.

Similar simple hand collating of blocks of punch-outs with the original data cards (binary) and changing of the MCR are all that are necessary to set up the deck for PLP or re-inverting the basis. Special short MCR's are usually accumulated for such things as: transforming a new right hand side,

computing and printing the d_j, altering a vector in the basis, etc. A special routine is provided for copying a $\eta_r^{i(t)}$ tape in reverse order so that a problem can be picked up after bad luck with tapes.

PART C - PRE-DESIGNED MODELS AND ALTERNATE METHODS

The objectives which have motivated the programs discussed in Parts A and B have been mainly three fold:

(i) to perform calculations for any linear programming model which might be presented, within current computing capacity, in as efficient and accurate a way as possible;

(ii) to accept any reasonable special requirements and even to anticipate them;

(iii) to increase the size of problems which are computationally feasible.

Even assuming a degree of success in all three, there is much left to be desired. As to (ii), it is obvious that new requests will continually be made which are unexpected. While a few special devices can be pre-designed, the main reliance must be on an extremely flexible code in which the algorithm can be modified in any particular without upsetting the whole apple-cart. We believe we are in a better position in this regard than in the other matters. Although it takes time to carefully plan the organization of a computer program, it pays off in the long run. The important thing is that the code truly reflect the basic method and not be a hodge-podge of standard library sub-routines assembled for expediency. This will be even more important for the higher levels of abstraction which will be required in some proposed methods. It is also important for the programmer to be on the look-out for situations amenable to simple tricks. For example, dualizing a problem sometimes renders special computations easier. Incidentally, we have not found the actual dual algorithm very useful in practice. It is slow and the

composite algorithm was developed mainly to replace its use.* To a lesser extent, the same is true of PLP.

It seems that (i) and (iii) are somewhat antithetical. Accuracy is usually not an important consideration in final results but is necessary to maintain the progress of the computations. Efficiency is a difficult thing to measure and depends on one's perspective, but for very large problems it can be taken as meaning lowest possible machine time. Hence, accuracy and efficiency are as much a part of (iii) as of (i). But, if the size of problems is to be increased efficiently, then special methods must be developed and most of the suggestions so far depend on structure in the a_j^i matrix. Thus it appears that we must begin to insist that, if problems are to exceed a certain size (and this has been the main demand), then they must fit into certain patterns. This may not be as restrictive as it sounds. It seems unlikely that many problems would be formulated by someone just writing down, say, 500 random equations as a set of constraints. Indeed, we anticipate that, as systems increase in size, there will be more requests for constructing the a_j^i matrix itself by machine. If this is so, then it virtually implies a pattern and there should be some degree of freedom in the way it is to be constructed.

The main difficulty at present, of course, is to devise a set of patterns which lead to efficient computing procedures and which, at the same, time encompass realistic problems. Work along these lines has barely begun. We will

---

*Markowitz has a straight-forward method of obtaining a new feasible solution after a change in the right hand side, provided the change is proportional in a positive sense to legitimate vectors in the system. This has not been used extensively but has much merit. It is simply the application of the theorem on basic solutions: If any feasible solution exists, then a basic feasible solution exists.

mention, however, one such pre-designed model which is now under development.

In numerous applications of linear programming, it has been observed that the a_j^i matrix can be partitioned into a special block triangular form. Extending our notation to elements which are themselves matrices, let

$$A_J^I = (a_j^i)_J^I \quad (i = 1, 2, \ldots, m_I;\ j = 1, 2, \ldots, n_J).$$

Then the structured matrix

$$\begin{matrix} A_0^0 & A_1^0 & A_2^0 & - - - & A_M^0 \\ & A_1^1 & & & \\ & & A_2^2 & & \\ & & & \ddots & \\ & & & & A_M^M \end{matrix} \qquad \text{(unspecified blocks all zero)}$$

gives rise to bases which, by a relatively simple transformation, can be maintained in a similar structured form. It is fairly clear that most of the numerical operations can be confined to the individual blocks, assuming the corresponding diagonal blocks of the basis have rank m_I, respectively. A great deal of study has gone into devising the proper sequence of operations to make the arithmetic as short as possible.

Examples of this structure occur in the transportation problem, the metal-working problem studied by Markowitz, the gasoline blending model of Charnes, Cooper and Mellon, and the two-period air-transport procurement problem of Manne. A special method based on this structure would not be economical in all instances but it is believed there is considerable potential demand for such a technique.

We cannot close without outlining another technique which has recently been used with spectacular results on small problems. Like the one above, it was proposed in an attempt to increase the size of problems which can be handled. In both cases, it turns out that these methods are more efficient for some problems well within the capacity of current codes than the standard methods. In the following, there is no special assumption on the a_j^i matrix except that it be sparse. The necessary degree of sparseness for efficiency is not yet known since it is heavily dependent on code logic.

In 1954, Markowitz pointed out that if it were possible to obtain a form of inverse which could be applied quickly and which was not too expensive to generate, one could invert a basis and then use the product form to modify it for several iterations. When the transformations became too long, then the latest basis could be re-inverted, there being some optimal point for doing this.

Now it is the number of multiplications and additions that determine the time for applying an inverse, in short the number of non-zero elements in the form of inverse used. It is theoretically possible to obtain a form of inverse in which the number of non-zero elements is very little greater than the number of non-zero elements in the original matrix. The product form has, in general, fewer non-zero elements than the full inverse of a sparse matrix. For example:

$$A^{-1} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1/2 & -1/2 & -1/2 & -1/2 \\ 1/2 & 1/2 & -1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 & 1/2 \end{bmatrix}$$

with 16 non-zero elements. However A^{-1} can be represented in product form by the four $\eta_r^{i(t)}$ vectors, where $r(t) = t$:

$$\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad \begin{bmatrix} -1/2 \\ -1/2 \\ -1/2 \\ 1/2 \end{bmatrix}$$

which have 10 non-zero elements. Note that if $r(1) = 4$, the number of elements in the product form changes. Thus, the product form does not, in general, minimize the number of non-zero elements required, in fact there is no known algorithm for doing so, at least in a practical sense. However, as Markowitz further pointed out, one does, in fact, do a pretty good job of it in solving a system of simultaneous equations by hand. If several coefficients in the equations are zero, one does not proceed to triangularize in a straightforward manner but rather seeks to eliminate first those rows and columns with the fewest elements.

His first proposal was to operate first on a grid or "picture" of the matrix, following through the effect of the eliminations on the zeros, without actually obtaining the numerical values of the transformed elements $\alpha^i_{h(t)}$. The result of this was to be an "agenda" consisting of a sequence of pairs of indices $(r(t), s(t))$, $t = 1, 2, \ldots, m$. The rule for choosing $r(t)$ and $s(t)$ was to be as follows:

> Let σ_h be the count of the non-zero elements remaining in column h at this point of the operations and let ρ_i be similarly the count for row i. Then, r and s are chosen by
>
> $$\min_{\alpha^i_{h(t)} \neq 0} \left\{ (\rho_i - 1)(\sigma_h - 1) \right\} = (\rho_r - 1)(\sigma_s - 1).$$

Having determined the agenda, the actual transformations were to be carried through forming "forward solutions" and " backward solutions" in the usual manner of eliminating. The forward transformations are like our $\eta_h^{i(t)}$ matrices and the backward transformations like the transpose of these.

There were several technical difficulties with coding up such a procedure that would produce a result compatible with the product form, and the bookkeeping was unusually difficult. There was also a theoretical difficulty. Suppose that in carrying out the agenda, one of the pivot elements $\partial^r_{s(t)}$ became zero by chance. What course of action should be taken and could anything be salvaged?

The project was shelved for several months and then re-investigated. It was found that the agenda and the actual transformations could be performed simultaneously, or rather, in alternating steps. If a pivot element turns out zero it is simply rejected and another one picked. Also, by slightly changing the procedure, the backward as well as the forward transformations can be put into identically the same form as the usual product form. There is a permutation occurring between the two but this is recorded simply by inserting the agenda at this point. The complete details must await a later paper.

A program for RAND's own machine, the JOHNNIAC, has now been in operation for a short time. It has given remarkably good results on problems up to 51 equations in some 400 variables. The code is limited to 127 equations but this figure is probably ambitious since the machine has no tapes at present. There are 4096 words of HSS and 12,288 words of drum storage. The inversion code works so well that we have designed the JOHNNIAC simplex code so that,

if a job is interrupted, the basis must be re-inverted before continuing. Although the machine is slower than the 701, this code is faster than the 701 codes, partly due to fixed point arithmetic (double-precision) and very short drum transfer times. However, on one problem, this form of inverse had only 107 per cent as many non-trivial elements as the given basis. This makes the inverse fast to apply, including the part already generated when generating the inverse itself, that is, even the generation of the inverse is speeded up by advantageous feedback of its own features.

It is too early to say how successful this scheme would be on, say, the 704 with a very large, but sparse, system, say 400 or 500 equations. It has been amazing to the writer that the tremendous amount of bookkeeping and "juggling" in the JOHNNIAC code actually pays off. However, this is in line with all our other experience, that one can afford to do almost anything to cut down the number of non-zero multipliers and addends recorded. There are still one or two minor points of difficulty to be cleared up in the method but the potential flexibility exceeds even that of the 704 codes in some respects. (The present JOHNNIAC codes are not in a polished form but are quite convenient to use, even so.)

There are thus at least four variations of the simplex method for general problems, any one of which might be most advantageous depending on the machine and the size of problems to be provided for.

The original simplex method: for fairly small machines (2000 words total capacity) and small problems, this uses a fixed amount of storage and a relatively simple code.

The revised simplex method with explicit inverse: for small

machines, the size of problem can be increased somewhat by keeping only the inverse in HSS and running the a^i_j matrix through on cards, for example, again and again. The code can be kept fairly simple and much flexibility attained.

The revised simplex method with product form of inverse: for elaborate, automatic set-ups, this method is serving well but has definite limitations for problem size, especially without tapes.

The revised method with product form combined with elimination form of inverse: extends the size of problems that can be run on a given machine although not suitable for small machines due to large amount of code. The main drawback is that its advantages do not take effect until a non-trivial basis is known.

It is probably a waste of time to attempt to run problems on a machine with less than 2000 words of storage. No matter what method is used, the necessary code will cut down available storage drastically and any attempt at packing information, or other such tricks, will only make the code longer at the further expense of data storage. A program for 15 or 20 order systems might be used for a while, but before long, the customers would want to run bigger problems.

We believe that the time has come for specialization in the field and that future efforts should be directed toward concentrated study of problems of particular types, both computationally and analytically. If the experience

and attempts related in the foregoing help in any way to avoid future pit-
falls, they will have served one of their purposes.

BIBLIOGRAPHY

(NOTE: The classifications are not always exact nor mutually exclusive.)

BIBLIOGRAPHICAL REFERENCE:

1. Riley, Vera and Robert Loring Allen, Interindustry Economic Studies, Operations Research Office, The Johns Hopkins Press, Baltimore 18, Maryland, May 1955.

GENERAL BACKGROUND ON INPUT-OUTPUT ANALYSIS, CONVEX SETS, SIMPLEX METHOD, ETC.:

2. Cowles Commission Monograph No.13, Activity Analysis of Production and Allocation, T. J. Koopmans, editor. New York, John Wiley and Sons, Inc., 1951.
3. Symposium on Linear Inequalities and Programming, Washington, D.C. June 1951, sponsored jointly by U.S. Air Force and National Bureau of Standards. Published in Air Force PROJECT SCOOP MANUAL No.10, 1 April 1952.

SPECIFIC BACKGROUND ON SIMPLEX METHOD:

4. Orchard-Hays, Wm., Background, Development, and Extensions of the Revised Simplex Method, The RAND Corp., Santa Monica, Calif., RM-1433, 30 April 1954.

THE ORIGINAL SIMPLEX METHOD: (See first Chapters XXI, XXII, XXIII of 2. above.)

5. Charnes, A. and W. W. Cooper and A. Henderson, An Introduction to Linear Programming, New York, John Wiley and Sons, Inc., 1953.
6. Eisemann, K., Linear Programming, Quarterly of Applied Mathematics, Vol. XIII, No.3, October 1955.

THE REVISED SIMPLEX METHOD, DUALITY THEOREMS, COMPUTATIONAL TECHNIQUES, SPECIAL APPLICATIONS, EXTENSIONS, ETC.:

7. Dantzig, G. B. and others, Notes on Linear Programming, The RAND Corp., Santa Monica, Calif. A growing series of Research Memoranda (now about 30 papers) referred to below as RAND L.P. followed by Roman numeral indicating part number. In particular here:

   a. G.B.D. with Alex Orden and Philip Wolfe, The Generalized Simplex Method for Minimizing a Linear Form under Linear Inequality Restraints, RAND L.P. I, RM-1264, 5 April 1954.

   b. G.B.D. and Orden, Duality Theorems, RAND L.P. II, RM-1265, 30 October 1953.

   c. G.B.D., Upper Bounds, Secondary Constraints, and Block Triangularity in Linear Programming, RAND L.P. VIII, IX, X (one paper), RM-1367, 26 April 1954.

   d. G.B.D., Linear Programming Under Uncertainty, RAND L.P. XVII, RM-1374, 8 March 1955.

   e. G.B.D., The Dual Simplex Algorithm, RAND L.P. VII, RM-1270, 3 July 1954.

8. Manne, A. S. and H. Markowitz, On the Solution of Discrete Programming Problems, The RAND Corp., P-711.

9. Lemke, C.E., The Dual Method for Solving the Linear Programming Problem, Carnegie Institute of Technology, Dept. of Math., Technical Report No.29, March 4, 1953.

10. Gass, Saul and Thomas Saaty, The Computational Algorithm for the Parametric Objective Function, Naval Research Logistics Quarterly, Vol. 2, Nos. 1-2, March-June 1955, pp. 39-45.

EXPERIENCE WITH THE SIMPLEX METHOD:

11. Hoffman, A. J. and M. Mannos, D. Sokolowsky, N.W.Wiegman, Computational Experience in Solving Linear Programs, National Bureau of Standards Report 2501, May 15, 1953. Also published in Journal of the Society for Industrial and Applied Mathematics, September 1953, Vol. 1, No.1.

12. Orden, Alex, Application of the Simplex Method to a Variety of Matrix Problems, pp.28-50 of Ref.3.

13. Hoffman, A.J., Cycling in the Simplex Algorithm, National Bureau of Standards Report, 2974, December 16, 1953.

14. Prinz, D. G., Some Experiences on the Manchester Computer with the Simplex Method, Ferranti Ltd., Moston, Manchester, 10, England, April 1954.

15. Orchard-Hays, Wm., Computational Experience in Solving Linear Programming Problems, The RAND Corp., P-482, March 1954.

RELATED METHODS:

16. Beale, E.M.L., An Alternative Method for Linear Programming, Proceedings of the Cambridge Philosophical Society, Vo.50, Part 3, July 1954.

17. Frisch, Ragnar, The Multiplex Method for Linear Programming, Memorandum fra Universitetets SocialØkonomiske Institutt, Oslo, 17 October 1955.

APPLICATIONS (Referred to in text):

18. Charnes, A. and W. W. Cooper and B. Mellon., Blending Aviation Gasolines, Econometrica, vol. 20, No.20, April 1952, pp. 135-159.

19. Markowitz, Harry, Process Analysis of the Metal Working Industries, The RAND Corp., RM-1085, 12 May 1953.

20. Manne, A. S., Scheduling of Petroleum Refinery Operations, Harvard University Press, 1956.

21. Manne, A. S., An Application of Linear Programming to the Procurement of Transport Aircraft, The RAND Corp., P-675A (abstract only), May 13, 1955.