Quasi-Newton methods like `dfpmin` work well with the approximate line minimization done by `lnsrch`. The routines `powell` (§10.5) and `frprmn` (§10.6), however, need more accurate line minimization, which is carried out by the routine `linmin`.

Advanced Implementations of Variable Metric Methods

Although rare, it can conceivably happen that roundoff errors cause the matrix $\mathbf{H}_i$ to become nearly singular or non-positive-definite. This can be serious, because the supposed search directions might then not lead downhill, and because nearly singular $\mathbf{H}_i$'s tend to give subsequent $\mathbf{H}_i$'s that are also nearly singular.

There is a simple fix for this rare problem, the same as was mentioned in §10.4: In case of any doubt, you should *restart* the algorithm at the claimed minimum point, and see if it goes anywhere. Simple, but not very elegant. Modern implementations of variable metric methods deal with the problem in a more sophisticated way.

Instead of building up an approximation to $\mathbf{A}^{-1}$, it is possible to build up an approximation of $\mathbf{A}$ itself. Then, instead of calculating the left-hand side of (10.7.4) directly, one solves the set of linear equations

$$\mathbf{A}\cdot(\mathbf{x}_m - \mathbf{x}_i) = -\nabla f(\mathbf{x}_i) \tag{10.7.11}$$

At first glance this seems like a bad idea, since solving (10.7.11) is a process of order N^3 — and anyway, how does this help the roundoff problem? The trick is not to store $\mathbf{A}$ but rather a triangular decomposition of $\mathbf{A}$, its *Cholesky decomposition* (cf. §2.9). The updating formula used for the Cholesky decomposition of $\mathbf{A}$ is of order N^2 and can be arranged to guarantee that the matrix remains positive definite and nonsingular, even in the presence of finite roundoff. This method is due to Gill and Murray [1,2].

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1, §§3–6 (by K. W. Brodlie). [2]

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), pp. 56ff. [3]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 467–468.

10.8 Linear Programming and the Simplex Method

The subject of *linear programming*, sometimes called *linear optimization*, concerns itself with the following problem: For N independent variables $x_1, \ldots, x_N$, *maximize* the function

$$z = a_{01}x_1 + a_{02}x_2 + \cdots + a_{0N}x_N \tag{10.8.1}$$

subject to the primary constraints

$$x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots \quad x_N \geq 0 \tag{10.8.2}$$

and simultaneously subject to $M = m_1 + m_2 + m_3$ additional constraints, m_1 of them of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{iN}x_N \le b_i \qquad (b_i \ge 0) \qquad i = 1, \ldots, m_1 \tag{10.8.3}$$

m_2 of them of the form

$$a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jN}x_N \ge b_j \ge 0 \qquad j = m_1 + 1, \ldots, m_1 + m_2 \tag{10.8.4}$$

and m_3 of them of the form

$$\begin{aligned} a_{k1}x_1 + a_{k2}x_2 + \cdots + a_{kN}x_N = b_k \ge 0 \\ k = m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 \end{aligned} \tag{10.8.5}$$

The various a_{ij}'s can have either sign, or be zero. The fact that the b's must all be nonnegative (as indicated by the final inequality in the above three equations) is a matter of convention only, since you can multiply any contrary inequality by -1. There is no particular significance in the number of constraints M being less than, equal to, or greater than the number of unknowns N.

A set of values $x_1 \ldots x_N$ that satisfies the constraints (10.8.2)–(10.8.5) is called a *feasible vector*. The function that we are trying to maximize is called the *objective function*. The feasible vector that maximizes the objective function is called the *optimal feasible vector*. An optimal feasible vector can fail to exist for two distinct reasons: (i) there are *no* feasible vectors, i.e., the given constraints are incompatible, or (ii) there is no maximum, i.e., there is a direction in N space where one or more of the variables can be taken to infinity while still satisfying the constraints, giving an unbounded value for the objective function.

As you see, the subject of linear programming is surrounded by notational and terminological thickets. Both of these thorny defenses are lovingly cultivated by a coterie of stern acolytes who have devoted themselves to the field. Actually, the basic ideas of linear programming are quite simple. Avoiding the shrubbery, we want to teach you the basics by means of a couple of specific examples; it should then be quite obvious how to generalize.

Why is linear programming so important? (i) Because "nonnegativity" is the usual constraint on any variable x_i that represents the tangible amount of some physical commodity, like guns, butter, dollars, units of vitamin E, food calories, kilowatt hours, mass, etc. Hence equation (10.8.2). (ii) Because one is often interested in additive (linear) limitations or bounds imposed by man or nature: minimum nutritional requirement, maximum affordable cost, maximum on available labor or capital, minimum tolerable level of voter approval, etc. Hence equations (10.8.3)–(10.8.5). (iii) Because the function that one wants to optimize may be linear, or else may at least be approximated by a linear function — since that is the problem that linear programming *can* solve. Hence equation (10.8.1). For a short, semipopular survey of linear programming applications, see Bland [1].

Here is a specific example of a problem in linear programming, which has $N = 4$, $m_1 = 2$, $m_2 = m_3 = 1$, hence $M = 4$:

$$\text{Maximize} \quad z = x_1 + x_2 + 3x_3 - \tfrac{1}{2}x_4 \tag{10.8.6}$$

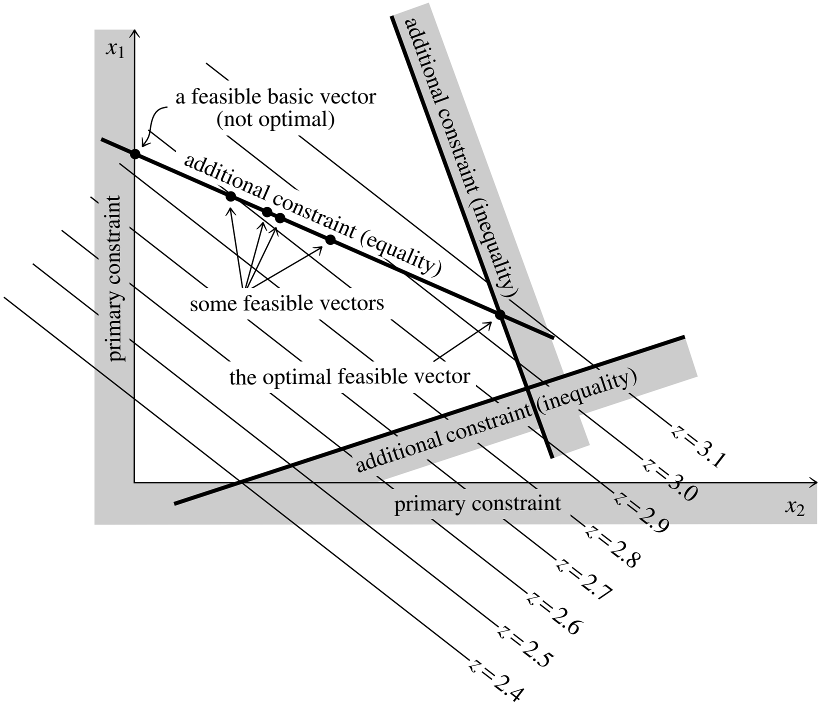

Figure 10.8.1. Basic concepts of linear programming. The case of only two independent variables, x_1, x_2, is shown. The linear function z, to be maximized, is represented by its contour lines. Primary constraints require x_1 and x_2 to be positive. Additional constraints may restrict the solution to regions (inequality constraints) or to surfaces of lower dimensionality (equality constraints). Feasible vectors satisfy all constraints. Feasible basic vectors also lie on the boundary of the allowed region. The simplex method steps among feasible basic vectors until the optimal feasible vector is found.

with all the x's nonnegative and also with

$$\begin{aligned} x_1 + 2x_3 &\le 740 \\ 2x_2 - 7x_4 &\le 0 \\ x_2 - x_3 + 2x_4 &\ge \tfrac{1}{2} \\ x_1 + x_2 + x_3 + x_4 &= 9 \end{aligned} \tag{10.8.7}$$

The answer turns out to be (to 2 decimals) $x_1 = 0$, $x_2 = 3.33$, $x_3 = 4.73$, $x_4 = 0.95$. In the rest of this section we will learn how this answer is obtained. Figure 10.8.1 summarizes some of the terminology thus far.

Fundamental Theorem of Linear Optimization

Imagine that we start with a full N-dimensional space of candidate vectors. Then (in mind's eye, at least) we carve away the regions that are eliminated in turn by each imposed constraint. Since the constraints are linear, every boundary introduced by this process is a plane, or rather hyperplane. Equality constraints of the form (10.8.5) force the feasible region onto hyperplanes of smaller dimension, while inequalities simply divide the then-feasible region into allowed and nonallowed pieces.

When all the constraints are imposed, either we are left with some feasible region or else there are no feasible vectors. Since the feasible region is bounded by hyperplanes, it is geometrically a kind of convex polyhedron or simplex (cf. §10.4). If there is a feasible region, can the optimal feasible vector be somewhere in its interior, away from the boundaries? No, because the objective function is linear. This means that it always has a nonzero vector gradient. This, in turn, means that we could always increase the objective function by running up the gradient until we hit a boundary wall.

The boundary of any geometrical region has one less dimension than its interior. Therefore, we can now run up the gradient projected into the boundary wall until we reach an edge of that wall. We can then run up that edge, and so on, down through whatever number of dimensions, until we finally arrive at a point, a *vertex* of the original simplex. Since this point has all N of its coordinates defined, it must be the solution of N simultaneous *equalities* drawn from the original set of equalities and inequalities (10.8.2)–(10.8.5).

Points that are feasible vectors and that satisfy N of the original constraints as equalities, are termed *feasible basic vectors*. If $N > M$, then a feasible basic vector has *at least* $N - M$ of its components equal to zero, since at least that many of the constraints (10.8.2) will be needed to make up the total of N. Put the other way, *at most* M components of a feasible basic vector are nonzero. In the example (10.8.6)–(10.8.7), you can check that the solution as given satisfies as equalities the last three constraints of (10.8.7) and the constraint $x_1 \geq 0$, for the required total of 4.

Put together the two preceding paragraphs and you have the *Fundamental Theorem of Linear Optimization*: If an optimal feasible vector exists, then there is a feasible basic vector that is optimal. (Didn't we warn you about the terminological thicket?)

The importance of the fundamental theorem is that it reduces the optimization problem to a "combinatorial" problem, that of determining which N constraints (out of the $M + N$ constraints in 10.8.2–10.8.5) should be satisfied by the optimal feasible vector. We have only to keep trying different combinations, and computing the objective function for each trial, until we find the best.

Doing this blindly would take halfway to forever. The *simplex method*, first published by Dantzig in 1948 (see [2]), is a way of organizing the procedure so that (i) a series of combinations is tried for which the objective function increases at each step, and (ii) the optimal feasible vector is reached after a number of iterations that is almost always no larger than of order M or N, whichever is larger. An interesting mathematical sidelight is that this second property, although known empirically ever since the simplex method was devised, was not proved to be true until the 1982 work of Stephen Smale. (For a contemporary account, see [3].)

Simplex Method for a Restricted Normal Form

A linear programming problem is said to be in *normal form* if it has no constraints in the form (10.8.3) or (10.8.4), but rather only equality constraints of the form (10.8.5) and nonnegativity constraints of the form (10.8.2).

For our purposes it will be useful to consider an even more restricted set of cases, with this additional property: Each equality constraint of the form (10.8.5) must have at least one variable that has a positive coefficient and *that appears uniquely in that one constraint only*. We can then choose one such variable in each constraint equation, and solve that constraint equation for it. The variables thus chosen are called *left-hand variables* or *basic variables*, and there are exactly M ($= m_3$) of them. The remaining $N - M$ variables are called *right-hand variables* or *nonbasic variables*. Obviously this *restricted normal form* can be achieved only in the case $M \leq N$, so that is the case that we will consider.

You may be thinking that our restricted normal form is so specialized that it is unlikely to include the linear programming problem that you wish to solve. Not at all! We will presently show how *any* linear programming problem can be transformed into restricted normal form. Therefore bear with us and learn how to apply the simplex method to a restricted normal form.

Here is an example of a problem in restricted normal form:

$$\text{Maximize} \quad z = 2x_2 - 4x_3 \tag{10.8.8}$$

with x_1, x_2, x_3, and x_4 all nonnegative and also with

$$\begin{aligned} x_1 &= 2 - 6x_2 + x_3 \\ x_4 &= 8 + 3x_2 - 4x_3 \end{aligned} \tag{10.8.9}$$

This example has $N = 4$, $M = 2$; the left-hand variables are x_1 and x_4; the right-hand variables are x_2 and x_3. The objective function (10.8.8) is written so as to depend only on right-hand variables; note, however, that this is not an actual restriction on objective functions in restricted normal form, since any left-hand variables appearing in the objective function could be eliminated algebraically by use of (10.8.9) or its analogs.

For any problem in restricted normal form, we can instantly read off a feasible basic vector (although not necessarily the *optimal* feasible basic vector). Simply set all right-hand variables equal to zero, and equation (10.8.9) then gives the values of the left-hand variables for which the constraints are satisfied. The idea of the simplex method is to proceed by a series of exchanges. In each exchange, a right-hand variable and a left-hand variable change places. At each stage we maintain a problem in restricted normal form that is equivalent to the original problem.

It is notationally convenient to record the information content of equations (10.8.8) and (10.8.9) in a so-called *tableau*, as follows:

		x_2	x_3
z	0	2	-4
x_1	2	-6	1
x_4	8	3	-4

(10.8.10)

You should study (10.8.10) to be sure that you understand where each entry comes from, and how to translate back and forth between the tableau and equation formats of a problem in restricted normal form.

The first step in the simplex method is to examine the top row of the tableau, which we will call the "z-row." Look at the entries in columns labeled by right-hand variables (we will call these "right-columns"). We want to imagine in turn the effect of increasing each right-hand variable from its present value of zero, while leaving all the other right-hand variables at zero. Will the objective function increase or decrease? The answer is given by the sign of the entry in the z-row. Since we want to increase the objective function, only right columns having positive z-row entries are of interest. In (10.8.10) there is only one such column, whose z-row entry is 2.

The second step is to examine the column entries below each z-row entry that was selected by step one. We want to ask how much we can increase the right-hand variable before one of the left-hand variables is driven negative, which is not allowed. If the tableau element at the intersection of the right-hand column and the left-hand variable's row is positive, then it poses no restriction: the corresponding left-hand variable will just be driven more and more positive. If *all* the entries in any right-hand column are positive, then there is no bound on the objective function and (having said so) we are done with the problem.

If one or more entries below a positive z-row entry are negative, then we have to figure out which such entry first limits the increase of that column's right-hand variable. Evidently the limiting increase is given by dividing the element in the right-hand column (which is called the *pivot element*) into the element in the "constant column" (leftmost column) of the pivot element's row. A value that is small in magnitude is most restrictive. The increase in the objective function for this choice of pivot element is then that value multiplied by the z-row entry of that column. We repeat this procedure on all possible right-hand columns to find the pivot element with the largest such increase. That completes our "choice of a pivot element."

In the above example, the only positive z-row entry is 2. There is only one negative entry below it, namely -6, so this is the pivot element. Its constant-column entry is 2. This pivot will therefore allow x_2 to be increased by $2 \div |6|$, which results in an increase of the objective function by an amount $(2 \times 2) \div |6|$.

The third step is to *do* the increase of the selected right-hand variable, thus making it a left-hand variable; and simultaneously to modify the left-hand variables, reducing the pivot-row element to zero and thus making it a right-hand variable. For our above example let's do this first by hand: We begin by solving the pivot-row equation for the new left-hand variable x_2 in favor of the old one x_1, namely

$$x_1 = 2 - 6x_2 + x_3 \qquad \rightarrow \qquad x_2 = \tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3 \tag{10.8.11}$$

We then substitute this into the old z-row,

$$z = 2x_2 - 4x_3 = 2\left[\tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3\right] - 4x_3 = \tfrac{2}{3} - \tfrac{1}{3}x_1 - \tfrac{11}{3}x_3 \tag{10.8.12}$$

and into all other left-variable rows, in this case only x_4,

$$x_4 = 8 + 3\left[\tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3\right] - 4x_3 = 9 - \tfrac{1}{2}x_1 - \tfrac{7}{2}x_3 \tag{10.8.13}$$

Equations (10.8.11)–(10.8.13) form the new tableau

$$
\begin{array}{c||c|c|c|}
 & & x_1 & x_3 \\
\hline
z & \frac{2}{3} & -\frac{1}{3} & -\frac{11}{3} \\
\hline
x_2 & \frac{1}{3} & -\frac{1}{6} & \frac{1}{6} \\
x_4 & 9 & -\frac{1}{2} & -\frac{7}{2} \\
\hline
\end{array}
\tag{10.8.14}
$$

The fourth step is to go back and repeat the first step, looking for another possible increase of the objective function. We do this as many times as possible, that is, until all the right-hand entries in the z-row are negative, signaling that no further increase is possible. In the present example, this already occurs in (10.8.14), so we are done.

The answer can now be read from the constant column of the final tableau. In (10.8.14) we see that the objective function is maximized to a value of $2/3$ for the solution vector $x_2 = 1/3$, $x_4 = 9$, $x_1 = x_3 = 0$.

Now look back over the procedure that led from (10.8.10) to (10.8.14). You will find that it could be summarized entirely in tableau format as a series of prescribed elementary matrix operations:

- Locate the pivot element and save it.
- Save the whole pivot column.
- Replace each row, except the pivot row, by that linear combination of itself and the pivot row which makes its pivot-column entry zero.
- Divide the pivot row by the negative of the pivot.
- Replace the pivot element by the reciprocal of its saved value.
- Replace the rest of the pivot column by its saved values divided by the saved pivot element.

This is the sequence of operations actually performed by a linear programming routine, such as the one that we will presently give.

You should now be able to solve almost any linear programming problem that starts in restricted normal form. The only special case that might stump you is if an entry in the constant column turns out to be zero at some stage, so that a left-hand variable is zero at the same time as all the right-hand variables are zero. This is called a *degenerate feasible vector*. To proceed, you may need to exchange the degenerate left-hand variable for one of the right-hand variables, perhaps even making several such exchanges.

Writing the General Problem in Restricted Normal Form

Here is a pleasant surprise. There exist a couple of clever tricks that render trivial the task of translating a general linear programming problem into restricted normal form!

First, we need to get rid of the inequalities of the form (10.8.3) or (10.8.4), for example, the first three constraints in (10.8.7). We do this by adding to the problem so-called *slack variables* which, when their nonnegativity is required, convert the inequalities to equalities. We will denote slack variables as y_i. There will be $m_1 + m_2$ of them. Once they are introduced, you treat them on an equal footing with the original variables x_i; then, at the very end, you simply ignore them.

For example, introducing slack variables leaves (10.8.6) unchanged but turns (10.8.7) into

$$\begin{aligned} x_1 + 2x_3 + y_1 &= 740 \\ 2x_2 - 7x_4 + y_2 &= 0 \\ x_2 - x_3 + 2x_4 - y_3 &= \tfrac{1}{2} \\ x_1 + x_2 + x_3 + x_4 &= 9 \end{aligned} \tag{10.8.15}$$

(Notice how the sign of the coefficient of the slack variable is determined by which sense of inequality it is replacing.)

Second, we need to insure that there is a set of M left-hand vectors, so that we can set up a starting tableau in restricted normal form. (In other words, we need to find a "feasible basic starting vector.") The trick is again to invent new variables! There are M of these, and they are called *artificial variables*; we denote them by z_i. You put exactly one artificial variable into each constraint equation on the following model for the example (10.8.15):

$$\begin{aligned} z_1 &= 740 - x_1 - 2x_3 - y_1 \\ z_2 &= -2x_2 + 7x_4 - y_2 \\ z_3 &= \tfrac{1}{2} - x_2 + x_3 - 2x_4 + y_3 \\ z_4 &= 9 - x_1 - x_2 - x_3 - x_4 \end{aligned} \tag{10.8.16}$$

Our example is now in restricted normal form.

Now you may object that (10.8.16) is not the same problem as (10.8.15) or (10.8.7) *unless all the z_i's are zero.* Right you are! There is some subtlety here! We must proceed to solve our problem in two phases. First phase: We replace our objective function (10.8.6) by a so-called *auxiliary objective function*

$$z' \equiv -z_1 - z_2 - z_3 - z_4 = -(749\tfrac{1}{2} - 2x_1 - 4x_2 - 2x_3 + 4x_4 - y_1 - y_2 + y_3) \tag{10.8.17}$$

(where the last equality follows from using 10.8.16). We now perform the simplex method on the auxiliary objective function (10.8.17) with the constraints (10.8.16). Obviously the auxiliary objective function will be maximized for nonnegative z_i's if all the z_i's are zero. We therefore expect the simplex method in this first phase to produce a set of left-hand variables drawn from the x_i's and y_i's only, with all the z_i's being right-hand variables. Aha! We then cross out the z_i's, leaving a problem involving only x_i's and y_i's in restricted normal form. In other words, the first phase produces an initial feasible basic vector. Second phase: Solve the problem produced by the first phase, using the original objective function, not the auxiliary.

And what if the first phase *doesn't* produce zero values for all the z_i's? That signals that there is *no* initial feasible basic vector, i.e., that the constraints given to us are inconsistent among themselves. Report that fact, and you are done.

Here is how to translate into tableau format the information needed for both the first and second phases of the overall method. As before, the underlying problem to be solved is as posed in equations (10.8.6)–(10.8.7).

		x_1	x_2	x_3	x_4	y_1	y_2	y_3
z	0	1	1	3	$-\frac{1}{2}$	0	0	0
z_1	740	-1	0	-2	0	-1	0	0
z_2	0	0	-2	0	7	0	-1	0
z_3	$\frac{1}{2}$	0	-1	1	-2	0	0	1
z_4	9	-1	-1	-1	-1	0	0	0
z'	$-749\frac{1}{2}$	2	4	2	-4	1	1	-1

(10.8.18)

This is not as daunting as it may, at first sight, appear. The table entries inside the box of double lines are no more than the coefficients of the original problem (10.8.6)–(10.8.7) organized into a tabular form. In fact, these entries, along with the values of N, M, m_1, m_2, and m_3, are the only input that is needed by the simplex method routine below. The columns under the slack variables y_i simply record whether each of the M constraints is of the form $\leq$, $\geq$, or $=$; this is redundant information with the values m_1, m_2, m_3, as long as we are sure to enter the rows of the tableau in the correct respective order. The coefficients of the auxiliary objective function (bottom row) are just the negatives of the column sums of the rows above, so these are easily calculated automatically.

The output from a simplex routine will be (i) a flag telling whether a finite solution, no solution, or an unbounded solution was found, and (ii) an updated tableau. The output tableau that derives from (10.8.18), given to two significant figures, is

		x_1	y_2	y_3	$\cdots$
z	17.03	$-.95$	$-.05$	-1.05	$\cdots$
x_2	3.33	$-.35$	$-.15$	$.35$	$\cdots$
x_3	4.73	$-.55$	$.05$	$-.45$	$\cdots$
x_4	$.95$	$-.10$	$.10$	$.10$	$\cdots$
y_1	730.55	$.10$	$-.10$	$.90$	$\cdots$

(10.8.19)

A little counting of the x_i's and y_i's will convince you that there are $M + 1$ rows (including the z-row) in both the input and the output tableaux, but that only $N + 1 - m_3$ columns of the output tableau (including the constant column) contain any useful information, the other columns belonging to now-discarded artificial variables. In the output, the first numerical column contains the solution vector, along with the maximum value of the objective function. Where a slack variable (y_i) appears on the left, the corresponding value is the amount by which its inequality is safely satisfied. Variables that are not left-hand variables in the output tableau have zero values. Slack variables with zero values represent constraints that are satisfied as equalities.

Routine Implementing the Simplex Method

The following routine is based algorithmically on the implementation of Kuenzi, Tzschach, and Zehnder [4]. Aside from input values of M, N, m_1, m_2, m_3, the principal input to the routine is a two-dimensional array a containing the portion of the tableau (10.8.18) that is contained between the double lines. This input occupies the $M + 1$ rows and $N + 1$ columns of a[1..m+1][1..n+1]. Note, however, that reference is made internally to row $M + 2$ of a (used for the auxiliary objective function, just as in 10.8.18). Therefore the variable declared as float **a, must point to allocated memory allowing references in the subrange

$$\texttt{a}[i][k], \quad i = 1 \ldots \texttt{m+2}, \; k = 1 \ldots \texttt{n+1} \tag{10.8.20}$$

You will suffer endless agonies if you fail to understand this simple point. Also do not neglect to order the rows of a in the same order as equations (10.8.1), (10.8.3), (10.8.4), and (10.8.5), that is, objective function, $\le$-constraints, $\ge$-constraints, $=$-constraints.

On output, the tableau a is indexed by two returned arrays of integers. iposv[j] contains, for j$= 1 \ldots M$, the number i whose original variable x_i is now represented by row j+1 of a. These are thus the left-hand variables in the solution. (The first row of a is of course the z-row.) A value $i > N$ indicates that the variable is a y_i rather than an x_i, $x_{N+j} \equiv y_j$. Likewise, izrov[j] contains, for j$= 1 \ldots N$, the number i whose original variable x_i is now a right-hand variable, represented by column j+1 of a. These variables are all zero in the solution. The meaning of $i > N$ is the same as above, except that $i > N + m_1 + m_2$ denotes an artificial or slack variable which was used only internally and should now be entirely ignored.

The flag icase is set to zero if a finite solution is found, $+1$ if the objective function is unbounded, -1 if no solution satisfies the given constraints.

The routine treats the case of degenerate feasible vectors, so don't worry about them. You may also wish to admire the fact that the routine does not require storage for the columns of the tableau (10.8.18) that are to the right of the double line; it keeps track of slack variables by more efficient bookkeeping.

Please note that, as given, the routine is only "semi-sophisticated" in its tests for convergence. While the routine properly implements tests for inequality with zero as tests against some small parameter EPS, it does not adjust this parameter to reflect the scale of the input data. This is adequate for many problems, where the input data do not differ from unity by too many orders of magnitude. If, however, you encounter endless cycling, then you should modify EPS in the routines simplx and simp2. Permuting your variables can also help. Finally, consult [5].

```
#include "nrutil.h"
#define EPS 1.0e-6
Here EPS is the absolute precision, which should be adjusted to the scale of your variables.
#define FREEALL free_ivector(l3,1,m);free_ivector(l1,1,n+1);

void simplx(float **a, int m, int n, int m1, int m2, int m3, int *icase,
    int izrov[], int iposv[])
Simplex method for linear programming. Input parameters a, m, n, mp, np, m1, m2, and m3,
and output parameters a, icase, izrov, and iposv are described above.
{
    void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
```

```
    float *bmax);
void simp2(float **a, int n, int l2[], int nl2, int *ip, int kp, float *q1);
void simp3(float **a, int i1, int k1, int ip, int kp);
int i,ip,is,k,kh,kp,nl1;
int *l1,*l3;
float q1,bmax;

if (m != (m1+m2+m3)) nrerror("Bad input constraint counts in simplx");
l1=ivector(1,n+1);
l3=ivector(1,m);
nl1=n;
for (k=1;k<=n;k++) l1[k]=izrov[k]=k;
```
Initialize index list of columns admissible for exchange, and make all variables initially right-hand.
```
for (i=1;i<=m;i++) {
    if (a[i+1][1] < 0.0) nrerror("Bad input tableau in simplx");
```
Constants b_i must be nonnegative.
```
    iposv[i]=n+i;
```
Initial left-hand variables. m1 type constraints are represented by having their slack variable initially left-hand, with no artificial variable. m2 type constraints have their slack variable initially left-hand, with a minus sign, and their artificial variable handled implicitly during their first exchange. m3 type constraints have their artificial variable initially left-hand.
```
}
if (m2+m3) {                                    Origin is not a feasible starting so-
    for (i=1;i<=m2;i++) l3[i]=1;                    lution: we must do phase one.
```
Initialize list of m2 constraints whose slack variables have never been exchanged out of the initial basis.
```
    for (k=1;k<=(n+1);k++) {                    Compute the auxiliary objective func-
        q1=0.0;                                     tion.
        for (i=m1+1;i<=m;i++) q1 += a[i+1][k];
        a[m+2][k] = -q1;
    }
    for (;;) {
        simp1(a,m+1,l1,nl1,0,&kp,&bmax);        Find max. coeff. of auxiliary objec-
        if (bmax <= EPS && a[m+2][1] < -EPS) {      tive fn.
            *icase = -1;
```
Auxiliary objective function is still negative and can't be improved, hence no feasible solution exists.
```
            FREEALL return;
        } else if (bmax <= EPS && a[m+2][1] <= EPS) {
```
Auxiliary objective function is zero and can't be improved; we have a feasible starting vector. Clean out the artificial variables corresponding to any remaining equality constraints by goto one and then move on to phase two.
```
            for (ip=m1+m2+1;ip<=m;ip++) {
                if (iposv[ip] == (ip+n)) {      Found an artificial variable for an
                    simp1(a,ip,l1,nl1,1,&kp,&bmax);     equality constraint.
                    if (bmax > EPS)             Exchange with column correspond-
                        goto one;                   ing to maximum pivot element
                }                                   in row.
            }
            for (i=m1+1;i<=m1+m2;i++)           Change sign of row for any m2 con-
                if (l3[i-m1] == 1)                  straints still present from the ini-
                    for (k=1;k<=n+1;k++)            tial basis.
                        a[i+1][k] = -a[i+1][k];
            break;                              Go to phase two.
        }
        simp2(a,m,n,&ip,kp);                    Locate a pivot element (phase one).
        if (ip == 0) {                          Maximum of auxiliary objective func-
            *icase = -1;                            tion is unbounded, so no feasi-
            FREEALL return;                         ble solution exists.
        }
one:    simp3(a,m+1,n,ip,kp);
```
Exchange a left- and a right-hand variable (phase one), then update lists.

```
            if (iposv[ip] >= (n+m1+m2+1)) {       Exchanged out an artificial variable
                for (k=1;k<=nl1;k++)                  for an equality constraint. Make
                    if (l1[k] == kp) break;           sure it stays out by removing it
                --nl1;                                from the l1 list.
                for (is=k;is<=nl1;is++) l1[is]=l1[is+1];
            } else {
                kh=iposv[ip]-m1-n;
                if (kh >= 1 && l3[kh]) {          Exchanged out an m2 type constraint
                    l3[kh]=0;                         for the first time. Correct the
                    ++a[m+2][kp+1];                   pivot column for the minus sign
                    for (i=1;i<=m+2;i++)              and the implicit artificial vari-
                        a[i][kp+1] = -a[i][kp+1];     able.
                }
            }
            is=izrov[kp];                         Update lists of left- and right-hand
            izrov[kp]=iposv[ip];                      variables.
            iposv[ip]=is;
        }                                         Still in phase one, go back to the
    }                                                 for(;;).
    End of phase one code for finding an initial feasible solution. Now, in phase two, optimize
    it.
    for (;;) {
        simp1(a,0,l1,nl1,0,&kp,&bmax);            Test the z-row for doneness.
        if (bmax <= EPS) {                        Done. Solution found. Return with
            *icase=0;                                 the good news.
            FREEALL return;
        }
        simp2(a,m,n,&ip,kp);                      Locate a pivot element (phase two).
        if (ip == 0) {                            Objective function is unbounded. Re-
            *icase=1;                                 port and return.
            FREEALL return;
        }
        simp3(a,m,n,ip,kp);                       Exchange a left- and a right-hand
        is=izrov[kp];                                 variable (phase two),
        izrov[kp]=iposv[ip];
        iposv[ip]=is;
    }                                             and return for another iteration.
}
```

The preceding routine makes use of the following utility functions.

```
#include <math.h>

void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
    float *bmax)
Determines the maximum of those elements whose index is contained in the supplied list ll,
either with or without taking the absolute value, as flagged by iabf.
{
    int k;
    float test;

    if (nll <= 0)                    No eligible columns.
        *bmax=0.0;
    else {
        *kp=ll[1];
        *bmax=a[mm+1][*kp+1];
        for (k=2;k<=nll;k++) {
            if (iabf == 0)
                test=a[mm+1][ll[k]+1]-(*bmax);
```

```
            else
                test=fabs(a[mm+1][ll[k]+1])-fabs(*bmax);
            if (test > 0.0) {
                *bmax=a[mm+1][ll[k]+1];
                *kp=ll[k];
            }
        }
    }
}
```

```
#define EPS 1.0e-6

void simp2(float **a, int m, int n, int *ip, int kp)
```
Locate a pivot element, taking degeneracy into account.
```
{
    int k,i;
    float qp,q0,q,q1;

    *ip=0;
    for (i=1;i<=m;i++)
        if (a[i+1][kp+1] < -EPS) break;        Any possible pivots?
    if (i>m) return;
    q1 = -a[i+1][1]/a[i+1][kp+1];
    *ip=i;
    for (i=*ip+1;i<=m;i++) {
        if (a[i+1][kp+1] < -EPS) {
            q = -a[i+1][1]/a[i+1][kp+1];
            if (q < q1) {
                *ip=i;
                q1=q;
            } else if (q == q1) {              We have a degeneracy.
                for (k=1;k<=n;k++) {
                    qp = -a[*ip+1][k+1]/a[*ip+1][kp+1];
                    q0 = -a[i+1][k+1]/a[i+1][kp+1];
                    if (q0 != qp) break;
                }
                if (q0 < qp) *ip=i;
            }
        }
    }
}
```

```
void simp3(float **a, int i1, int k1, int ip, int kp)
```
Matrix operations to exchange a left-hand and right-hand variable (see text).
```
{
    int kk,ii;
    float piv;

    piv=1.0/a[ip+1][kp+1];
    for (ii=1;ii<=i1+1;ii++)
        if (ii-1 != ip) {
            a[ii][kp+1] *= piv;
            for (kk=1;kk<=k1+1;kk++)
                if (kk-1 != kp)
                    a[ii][kk] -= a[ip+1][kk]*a[ii][kp+1];
        }
```

```
    for (kk=1;kk<=k1+1;kk++)
        if (kk-1 != kp) a[ip+1][kk] *= -piv;
    a[ip+1][kp+1]=piv;
}
```

Other Topics Briefly Mentioned

Every linear programming problem in normal form with N variables and M constraints has a corresponding *dual* problem with M variables and N constraints. The tableau of the dual problem is, in essence, the transpose of the tableau of the original (sometimes called *primal*) problem. It is possible to go from a solution of the dual to a solution of the primal. This can occasionally be computationally useful, but generally it is no big deal.

The *revised simplex method* is exactly equivalent to the simplex method in its choice of which left-hand and right-hand variables are exchanged. Its computational effort is not significantly less than that of the simplex method. It does differ in the organization of its storage, requiring only a matrix of size $M \times M$, rather than $M \times N$, in its intermediate stages. If you have a lot of constraints, and memory size is one of them, then you should look into it.

The *primal-dual algorithm* and the *composite simplex algorithm* are two different methods for avoiding the two phases of the usual simplex method: Progress is made simultaneously towards finding a feasible solution and finding an optimal solution. There seems to be no clearcut evidence that these methods are superior to the usual method by any factor substantially larger than the "tender-loving-care factor" (which reflects the programming effort of the proponents).

Problems where the objective function and/or one or more of the constraints are replaced by expressions nonlinear in the variables are called *nonlinear programming problems*. The literature on such problems is vast, but outside our scope. The special case of quadratic expressions is called *quadratic programming*. Optimization problems where the variables take on only integer values are called *integer programming* problems, a special case of *discrete optimization* generally. The next section looks at a particular kind of discrete optimization problem.

CITED REFERENCES AND FURTHER READING:

Bland, R.G. 1981, *Scientific American*, vol. 244 (June), pp. 126–144. [1]

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press). [2]

Kolata, G. 1982, *Science*, vol. 217, p. 39. [3]

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), Chapters 7–8.

Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders).

Gass, S.T. 1969, *Linear Programming*, 3rd ed. (New York: McGraw-Hill).

Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley).

Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience).

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press). [4]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.10.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [5]

10.9 Simulated Annealing Methods

The *method of simulated annealing* [1,2] is a technique that has attracted significant attention as suitable for optimization problems of large scale, especially ones where a desired global extremum is hidden among many, poorer, local extrema. For practical purposes, simulated annealing has effectively "solved" the famous *traveling salesman problem* of finding the shortest cyclical itinerary for a traveling salesman who must visit each of N cities in turn. (Other practical methods have also been found.) The method has also been used successfully for designing complex integrated circuits: The arrangement of several hundred thousand circuit elements on a tiny silicon substrate is optimized so as to minimize interference among their connecting wires [3,4]. Surprisingly, the implementation of the algorithm is relatively simple.

Notice that the two applications cited are both examples of *combinatorial minimization*. There is an objective function to be minimized, as usual; but the space over which that function is defined is not simply the N-dimensional space of N continuously variable parameters. Rather, it is a discrete, but very large, configuration space, like the set of possible orders of cities, or the set of possible allocations of silicon "real estate" blocks to circuit elements. The number of elements in the configuration space is factorially large, so that they cannot be explored exhaustively. Furthermore, since the set is discrete, we are deprived of any notion of "continuing downhill in a favorable direction." The concept of "direction" may not have any meaning in the configuration space.

Below, we will also discuss how to use simulated annealing methods for spaces with continuous control parameters, like those of §§10.4–10.7. This application is actually more complicated than the combinatorial one, since the familiar problem of "long, narrow valleys" again asserts itself. Simulated annealing, as we will see, tries "random" steps; but in a long, narrow valley, almost all random steps are uphill! Some additional finesse is therefore required.

At the heart of the method of simulated annealing is an analogy with thermodynamics, specifically with the way that liquids freeze and crystallize, or metals cool and anneal. At high temperatures, the molecules of a liquid move freely with respect to one another. If the liquid is cooled slowly, thermal mobility is lost. The atoms are often able to line themselves up and form a pure crystal that is completely ordered over a distance up to billions of times the size of an individual atom in all directions. This crystal is the state of minimum energy for this system. The amazing fact is that, for slowly cooled systems, nature is able to find this minimum energy state. In fact, if a liquid metal is cooled quickly or "quenched," it does not reach this state but rather ends up in a polycrystalline or amorphous state having somewhat higher energy.

So the essence of the process is *slow* cooling, allowing ample time for redistribution of the atoms as they lose mobility. This is the technical definition of *annealing*, and it is essential for ensuring that a low energy state will be achieved.