Preface

Most applied optimization problems involve constraints: What is the maximum profit that a manufacturer can make given a limited number of machines and a limited labour force? What is the minimum amount of fuel that a fleet of trucks can consume while making a specified set of deliveries? What is the smallest amount of silicon needed to etch an electronic circuit while respecting limits on signal propagation time, inter-wire distance, etc.? Applications of constrained optimization are everywhere in industry, business, and government.

Of course, the solution returned by an optimization algorithm must also be feasible: we want the best possible value of the objective function that satisfies all constraints and variable bounds. Some optimization algorithms are not even able to proceed towards optimality until a feasible solution is available. In addition, the optimization question can be converted to a feasibility question, and vice versa. And what happens when an algorithm is unable to find a feasible solution? How do we know what went wrong? How do we repair the model? Questions of optimization, feasibility, and infeasibility are inextricably linked.

There has been a surge of important developments related to feasibility and infeasibility in optimization in the last two decades, a trend that continues to accelerate even today. New and more efficient methods for seeking feasibility in difficult optimization forms such as mixed-integer programs and nonlinear programs are emerging. The first effective algorithms for analyzing infeasible models have been discovered and implemented in commercial software. A community of researchers in constraint programming has begun to integrate their knowledge and approaches with the optimization community. Unanticipated spin-off applications of the new algorithms are being found. It's an exciting time.

The goal of this book is to summarize the state of the art in recent work at the interface of optimization and feasibility. It should serve as a useful reference for researchers, graduate students, and software developers working on optimization, feasibility, infeasibility, and related topics. Readers having a reasonable grounding in optimization (linear and nonlinear programming, mixed-integer programming, etc.) should have no difficulty following the material.

Lightweight coverage of topics in constraint programming, with an emphasis on constraint satisfaction problems, is included to illustrate the extensive overlap and convergence in the two literatures. An ideal version of the book would cover topics in constraint programming in the same depth as topics in optimization, but this is beyond the scope of this project: collecting and organizing the wealth of new developments relating to feasibility and infeasibility in optimization. I hope the resulting book is useful to both optimizers and constraint programmers, and

that it helps accelerate the ongoing merger of the two communities merge into a stronger hybrid.

Acknowledgements

My graduate work was conducted during the late 1970s and early 1980s. Inspired by the energy crises of those times, I constructed network optimization models to minimize the use of energy in large industrial plants. Later I found the optimization modeling more interesting than the energy aspects of this work. I had noticed that some of the processing network models that I was using in the energy work suffered from an inability to carry flow in some of the arcs, a pathology later labeled *nonviability* (see Sec. 9.2). I developed algorithms to automatically identify and analyze this problem.

Enter Harvey Greenberg. At that time he was involved in a project to develop an *Intelligent Mathematical Programming System* (IMPS) (see e.g. Greenberg (1996b)), and consequently had an interest in algorithms for analyzing modeling errors of various types, such as nonviability. Harvey organized an extraordinary series of meetings on the IMPS topic for an eclectic group of researchers from academia and industry. Harvey invited me to one of these meetings and, as they say, the rest is history. Sitting in the bar one night after the IMPS meeting we had a discussion about whether or not you could isolate the cause of infeasibility in a linear program to an irreducible subset of the constraints defining the model. At the time, Harvey didn't think it could be done, but I did, so I bet him a beer that I could find a way to do so. As you will see in Part II of the book, I won that bet.

But there is a postscript to this story. I have now known Harvey for around twenty years, and we have gone on to make numerous one-beer bets on other issues in optimization. I have not won a single one of those subsequent bets, so I am currently several hundred beers in debt to him. But I have an even bigger debt than that. Harvey became my unofficial mentor, always ready to provide advice and suggestions and listen to my ideas. His influence on my work has been profound.

Harvey and Pascal Van Hentenryck both took the time to read an early draft of the book and provide advice and suggestions that greatly improved it. Both pointed out topics that should be greatly expanded upon, especially the material on constraint programming, but time is unfortunately limited, so a full treatment of that topic remains another project. And as clever as those two fellows are, I'm sure I've managed to hide a few errors in the manuscript that they did not find: those are mine alone.

Last but not least are the two incredible ladies in my life, my wife Linda and daughter Annie, who can finally look at this book and see what kept me glued to the computer for such long hours over the past year. Thanks for being there.

John W. Chinneck

2 Seeking Feasibility in Linear Programs

A general linear program has the form {min, max} $\boldsymbol{cx}$, subject to $\boldsymbol{Ax}$ {≤, ≥, =} $\boldsymbol{b}$, $\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}$, where $\boldsymbol{c}$ is a $1 \times n$ row vector, $\boldsymbol{x}$, $\boldsymbol{l}$, $\boldsymbol{u}$, and $\boldsymbol{b}$ are $n \times 1$ column vectors, and $\boldsymbol{A}$ is an $m \times n$ array, all consisting of real numbers. It is simple to find an immediate feasible solution for certain linear programs. For example, the origin (all variables equal zero) is always a basic feasible solution for an LP in a variation of canonical form that consists entirely of ≤ inequalities in which every element of $\boldsymbol{b}$ is nonnegative, and all variables are nonnegative. Similarly, network LPs in which the arc flow lower bounds are all zero admit the origin as a feasible solution.

It is more difficult to find a first feasible solution when the general LP is not in this special form, e.g. includes equality or ≥ constraints, or has negative entries in $\boldsymbol{b}$. In these cases, the origin is no longer available as a feasible solution, so more advanced methods of seeking feasibility are needed. In the simplex method, the most popular technique for reaching feasibility for general LPs is the two-phase method for reasons of numerical stability. The Big-*M* method, commonly presented in textbooks, is seldom used in implemented solvers.

More recently, infeasible-path interior point methods have been developed that do not necessarily reach feasibility until they also reach optimality. These techniques are beyond the scope of this book. See Wright (1997).

While reaching feasibility in LPs may seem to be a well-understood problem, there are a variety of heuristics which can speed the process considerably, such as crash starts, warm starts, and crossover from an infeasible solution.

2.1 The Phase 1 Algorithm

Given a basic feasible solution, the simplex algorithm is efficient at moving to a better adjacent basic feasible solution. It simply repeats this operation until recognizing that no further improvement is possible, and returns this final basic feasible solution as the optimum solution. As mentioned above, the difficulty with general LPs is that no basic feasible solution is immediately obvious except in very special cases. The phase 1 method addresses this problem by introducing nonnegative artificial variables into the problem so that a basic feasible solution is immediately available at the origin in the artificial space. A phase 1 objective function is also introduced which reaches its optimum value when the artificial variables are

driven to their lowest possible values; if all artifical variables achieve a value of zero, then we are at feasible solution for the original problem Details follow.

Without loss of generality, let us initially assume an LP in which all variables are restricted to be nonnegative, and all of the elements of $\boldsymbol{b}$ have nonnegative values. With these restrictions, the constraints that eliminate the origin as a basic feasible solution are the equality constraints and $\geq$ constraints that have strictly positive entries in $\boldsymbol{b}$. To permit the origin as a feasible point, we introduce a nonnegative artificial variable y_i for each such nonstandard constraint i, as follows (i) the inequality $\boldsymbol{a}_i \boldsymbol{x} \geq b_i$ is replaced by $\boldsymbol{a}_i \boldsymbol{x} + y_i \geq b_i$, and (ii) the equality $\boldsymbol{a}_i \boldsymbol{x} = b_i$ is replaced by $\boldsymbol{a}_i \boldsymbol{x} + y_i = b_i$. The phase 1 objective function is to minimize $W = \Sigma y_i$. The origin is a basic feasible solution for this phase 1 LP, hence the simplex method is able to initialize and iterate towards an optimum solution.

If the phase 1 LP terminates at an optimum solution in which $W = 0$, then it has found a point at which the artificial variables can be dropped and all of the original constraints are satisfied, i.e. a feasible point for the original problem. The solution process now initiates phase 2 at the current point by dropping all of the artificial variables and the phase 1 objective function, and re-introducing the original objective function. Ordinary simplex iterations then proceed to the optimum of the original objective function. Note that for efficiency reasons, the original objective function is normally included in the phase 1 matrix and updated as a nonbinding row so that it is in proper form when it comes time to solve the phase 2 problem.

On the other hand, if the phase 1 LP terminates at an optimum solution in which $W > 0$, then we know that the original LP is infeasible. W represents the sum of the violations of the equality and $\geq$ constraints, hence the size of W at the optimum solution is in some sense a measure of the size of the infeasibility. This notion can be generalized if the LP is fully elasticized (see Sec. 6.1.4). Other properties of the phase 1 solution, such as the dual prices of the slack variables, are useful in analyzing the cause of the infeasibility, as explained in later chapters.

There are some minor potential difficulties if the phase 1 solution terminates with $W = 0$ but the solution is degenerate. In this case, an artificial variable may have a value of zero and yet be in the basis. This can happen when the model has redundancies. However this is easy to recognize and handle. Dantzig and Thapa (1997, pp. 81– 82) list three ways to handle this problem, the simplest of which is to simply pivot the artificial variable out of the basis. This is done by choosing a nonzero element in a column for an original variable in the row for which the artificial variable is basic, and performing the pivot.

Note that it is possible to formulate a phase 1 that includes only a single artificial variable, however for implementation reasons this variant is not used in practice. See Nazareth (1987, pp. 147–149) for details.

2.2 The Big-*M* Method

The Big-*M* method requires the introduction of the same artificial variables as in the phase 1 method described above. The difference lies in how the artificial variables are driven out of the basis. "Big-*M*" refers to a large positive multiplier *M*, which is used as a penalty to discourage the inclusion of any artificial variables in the basis. The method works towards feasibility and optimality simultaneously within a single phase by using an appropriate form of the objective function:

- for maximization: $max\ Z = \boldsymbol{cx} - M\boldsymbol{y}$,
- for minimization: $min\ Z = \boldsymbol{cx} + M\boldsymbol{y}$.

As for the two-phase method described above, feasibility is recognized when all of the artificial variables are driven to zero. This may not happen until optimality is also reached.

The practical difficulty with the Big-*M* approach is that the large multiplier introduces numerical difficulties in the solution by dominating the calculations, however if the value of *M* is too small, then the procedure will terminate with an infeasible optimum solution. See Padberg (1999) for guidelines on choosing a suitable value for Big-*M*. Because of the numerical difficulties, the Big-*M* method is seldom used in practice.

2.3 Phase 1 from Any Basis

The phase 1 procedure given in Sec. 2.1 must start at the origin. A procedure that can be invoked from any given starting basis is preferable since it can be invoked when feasibility is lost (e.g. by accumulated rounding errors, or by changes to the model after it has been solved). As shown by Nazareth (1987), such a phase 1 procedure is possible if the upper and lower bounds on the variables are specifically considered (though this method applies equally well to singly-bounded or unbounded variables).

Consider the usual equation format of the LP after any necessary slack and surplus variables have been added: $\boldsymbol{Ax} = \boldsymbol{b}$. Partitioning the variables into the set of basic variables $\boldsymbol{x}_B^0$ and the set of nonbasic variables $\boldsymbol{x}_N^0$ at a given basis induces a similar partitioning of the $\boldsymbol{A}$ matrix into $\boldsymbol{B}^0$, the columns associated with the basic variables, and $\boldsymbol{N}^0$, the columns associated with the nonbasic variables. The rewritten LP equation is then

$$\boldsymbol{B}^0\boldsymbol{x}_B^0 + \boldsymbol{N}^0\boldsymbol{x}_N^0 = \boldsymbol{b}.$$

Now the following relationship holds at any iteration:

$$\boldsymbol{B}^0\boldsymbol{x}_B^0 = \boldsymbol{b} - \boldsymbol{N}^0\boldsymbol{x}_N^0.$$

The following phase 1 procedure considers that a variable can be nonbasic at either its upper or lower bound.

Given a basis, the values of the nonbasic variables are known (each nonbasic variable is at one of its bounds), and so all of the constant and variable values on

the right hand side of $\boldsymbol{B}^0\boldsymbol{x}_B^0 = \boldsymbol{b} - \boldsymbol{N}^0\boldsymbol{x}_N^0$ are known. Now we can solve for the values of the basic variables:

$$\boldsymbol{x}_B^0 = (\boldsymbol{B}^0)^{-1}(\boldsymbol{b} - \boldsymbol{N}^0\boldsymbol{x}_N^0).$$

Note that it may be numerically convenient to peg some of the nonbasic variables at values between their bounds; these variables are called *superbasic* (see (Nazareth 1987)). After solving for the values of the basic variables, some of them may be outside of their bounds during phase 1, i.e. the solution may be infeasible. The goal of the phase 1 procedure is then to drive all of the basic variables that are currently outside their bounds to within them.

Let us define $\overline{V}$ as the set of basic variables that violate their upper bounds, and $\underline{V}$ as the set of basic variables that violate their lower bounds. Nazareth (1987) shows that if the prices and reduced costs (π) are set to $c_i = 1$ if $i \in \overline{V}$, $c_i = -1$ if $i \in \underline{V}$, and $c = 0$ otherwise, then the sum of the infeasibilities is given correctly and the phase 1 reduced costs reflect the rate of change of the sum of the infeasibilities when a nonbasic variable is introduced into the infeasible basis $\boldsymbol{B}^0$. This means that whenever infeasibility is discovered, the cost vector $\boldsymbol{c}$ is replaced by the vector just described, and the simplex method is able to iterate in the normal manner towards feasibility. The cost component is reset to zero when a variable that is outside its bounds eventually satisfies them.

When variables can violate their bounds, or can be nonbasic at either the upper or the lower bound, there are several conditions to consider when choosing the leaving basic variable during simplex iterations (see Greenberg (1978)):

- A variable may be basic, outside its bounds and moving away from them, and hence will never be chosen as the leaving basic variable.
- A variable may be basic, outside its bounds and moving towards them, in which case it may pass through the violated bound and become nonbasic at the opposite bound.
- A variable may be basic and within its bounds, in which case it may become nonbasic at the first bound it meets.

These conditions are checked when determining the leaving basic variable, and the basic variable that most restricts the change in the value of the entering basic variable is chosen as the leaving basic variable, as usual. Note that an entering basic variable may be decreasing in value. Any variables that satisfy their bounds are kept inside their bounds by this procedure, while variables that violate their bounds are gradually made to satisfy them. In other words, the *number of infeasibilities* (NINF) is gradually reduced, eventually to zero if the LP is feasible.

While this procedure is effective, the fact that it keeps a variable within its bounds once it satisfies them can be overly restrictive. In some cases it is preferable to allow an entering basic variable to increase beyond the point at which the first currently-feasible basic variable encounters a bound because the overall sum of infeasibilities is still decreasing. When choosing the leaving basic variable, there are up to two thresholds associated with every basic variable:

- No thresholds if the basic variable is currently outside its bounds and moving away from them.

- One threshold if the basic variable is currently within its bounds. Beyond this threshold, the variable contributes to the sum of the infeasibilities.
- Two threshholds if the basic variable is currently outside its bounds and moving towards them. The first threshhold moves the basic variable into its feasible range, but is not blocking; beyond this threshold the variable no longer contributes to the sum of the infeasibilities. The second threshold is at the second bound and beyond this point the variable again contributes to the sum of the infeasibilities.

A more advanced procedure for choosing the leaving basic variable first sorts all of the thresholds in order from smallest to largest. It then looks at the rate of change of the sum of the infeasibilities in the zone between each threshold. The threshold dividing the last zone that shows a rate of decrease in the sum of the infeasibilities from the first zone that shows a rate of increase in the sum of the infeasibilities identifies the leaving basic variable. This emphasizes the decrease in the sum of the infeasibilities at the possible expense of increasing NINF. It is also possible to combine the two goals by examining the thresholds to reduce the sum of the infeasibilities as much as possible while not increasing NINF. This is done by choosing the threshold that is latest in the sorted list that does not increase NINF. Note that while you may pass through a threshold that causes a currently feasible basic variable to violate its bounds, a later threshold may cause a variable that currently violates its bounds to satisfy them, hence there is no net impact on NINF.

Nazareth (1987) describes the practical details of an efficient implementation of this scheme, including ways to immediately eliminate variables from consideration as the leaving basic variable, and ways to combine the calculations into a single pass through the candidate variables.

2.4 Crash Start Heuristics

A *crash start* in the context of linear programming is a procedure for generating a high quality initial basis. It may not be feasible, but it should be as close to feasibility as possible and have other helpful characteristics such as providing a nearly triangular matrix (which speeds the calculations). An LP with m independent rows and n original variables is normally converted to a form having $n + m$ variables, where one slack variable is added for each row. The main operation in crashing the initial basis is selecting m of the variables to be in the initial basis.

Once the basis is selected, the current values of the basic variables can be calculated. Then an appropriate phase 1 cost structure is assigned, as described in Sec. 2.3 and the phase 1 procedure iterates to feasibility.

The FortMP software (Ellison et al. 1999) describes a fairly standard crash procedure. The *unit basis* consisting of the slack variables is first set up, and then nonbasic original variables are gradually exchanged for basic slack variables. A basic slack variable is a candidate for an exchange with a nonbasic original variable if the pivot element at the intersection of the row for the basic slack variable

and the column for the nonbasic original variable is nonzero. To avoid the work involved in updating the matrix to check this condition, various heuristics are applied, using the fact that there has been no update to the pivot element if the variable columns selected in previous exchanges have nothing but zeroes on the current pivot row. If the rows and columns in the revised basis are ordered in the same order as their selection into the basis, this leads to a triangular basis.

The row selected for an exchange should have as few nonzero elements as possible in columns that are candidates for exchange into the basis, on the principle described above. A variable is then selected for exchange into the basis, and all other candidate nonbasic variables that have a nonzero pivot element in the current row are marked as unsuitable for exchange into the basis later (because, if selected, a matrix update would be required).

There are many ties for the selection of the row corresponding to the basic variable and the column corresponding to the nonbasic variable to be exchanged when the selection is based on sparsity as described. According to Ellison et al. (1999), the way in which ties are broken has a big impact on the feasibility of the final basis. Ties for the basic variable row are broken in favour of equality constraints (so that artificial variables are removed from the basis), and after that according to the degree of restriction, from most to least (i.e. basic variables that have a smaller range are exchanged first). Rows having free variables are never selected. Ties for the nonbasic variable column are broken by preferring to exchange variables that have the largest range, with first consideration being given to free variables (those without bounds). Fixed columns are never selected for exchange into the basis. The crash procedure can also be adjusted, primarily by changing the tie-breaking rules, to reduce the amount of degeneracy in the crashed basis.

If the phase 1 procedure uses artificial variables, then the crashing procedure can be designed to reduce the number of artificial variables in the basis. Only rows corresponding to basic artificial variables can be selected. The nonbasic variable is chosen so that the pivot element is of reasonable size; this helps avoid basis singularity. In this same vein, most solvers include a parameter that allows the user to select a minimum size for any pivot, usually set as a minimum fraction of the largest element in the column.

2.5 Crossover from an Infeasible Basis

Crossover normally refers to the process of moving from a feasible point provided by an interior point LP algorithm to a nearby feasible basis (the basic solution is desirable because it gives access to sensitivity analysis, etc.). However, if an advanced infeasible basis can be provided, e.g. by a crash procedure, then it is sometimes possible to crossover from that basis to a nearby feasible basis. This opens the possibility of using heuristic methods to generate an initial solution that is reasonably close to feasibility and then crossing over to a nearby feasible basis. The FortMP software (Ellison et al. 1999) includes techniques for providing a close-to-feasible initial point and for the subsequent crossover.

At a basis provided by a crash start, the solution is likely to include a certain number of superbasic variables (nonbasic variables that are not equal to one of their bounds, but instead lie between their bounds). So-called *purify* or *push* algorithms are then used to move superbasic variables to either a basic or nonbasic status, i.e. to arrive at a feasible basis. In FortMP (Ellison et al. 1999) there are separate push algorithms to remove primal superbasic variables and to remove dual superbasic variables. Both function in essentially the same way. The main idea is to examine the effect on the basic variables when the value of a superbasic variable is adjusted (in a manner similar to examining the effect of an entering basic variable on the existing basic variables). If the superbasic variable reaches one of its bounds before any basic variable does, then the superbasic is simply switched to nonbasic status. If a basic variable reaches one its bounds before the superbasic does, then a basis change is made, in which the basic variable is made nonbasic and the superbasic is made basic.

The version of the crash heuristics that tries to eliminate artificial variables is preferred for use with the push heuristics since it helps reduce the amount of work during the push phase. In addition, during the push phase, any original variables that are at their bounds after the crash are temporarily fixed at those values.

An approximate solution that is even closer to feasibility can be supplied by improving the output of the crash step before purifying. FortMP uses a *successive overrelaxation* (SOR) algorithm (Press et al. 1992), an iterative technique for solving systems of linear equations (see Sec. 2.8), for this purpose. The overall procedure has three steps: (i) apply the crash heuristic to create an approximately lower triangular basis, (ii) apply the successive overrelaxation algorithm to improve the point provided by the crash heuristic, and (iii) apply the push algorithms to cross over to a feasible basis. With luck the SOR procedure produces a feasible solution directly, which eases the crossover to a basic solution. If it does not produce a feasible solution, then the push algorithms may yet do so, though this is not guaranteed.

2.6 Advanced Starts: Hot and Warm Starts

If the LP solution process is stopped for any reason, the current basis and associated information may be stored. If the solution process is restarted later, this stored information provides a *hot start* which allows the solver to begin where it left off without repeating the set of iterations, including the phase 1 feasibility-seeking iterations, which originally generated the stored basis.

It frequently happens that minor changes are made to the LP model before it is restarted. This may happen because the conditions being modeled have changed, but it is an essential part of two important procedures. In solving mixed-integer programs via branch and bound, numerous LPs are solved in a tree-structured search for a solution that is both LP-feasible and integer-feasible. Each LP is identical to a previous LP except that a bound on one variable has been adjusted so

that the previous LP solution is rendered infeasible. In LP infeasibility analysis (see Sec. 6.2), several algorithms require the solution of sequences of LPs that differ by the addition or removal of one or several of the constraints or bounds. In cases such as these where the next LP to be solved is substantially similar (but not identical) to a previous LP, then a *warm start* that makes use of the previous solution and basis may be effective. This usually means that you can arrive at a new feasible (and optimal) solution in only a few iterations.

In warm-starting, if the changes made to the model have not rendered the warm-start point infeasible, then the primal simplex iterations just pick up where they left off and continue iterating to optimality. However, if the changes to the model have made the warm-start point primal-infeasible (normally by a change to a constraint or bound, or by the addition of one), then the warm-start point will still be dual feasible. The solver then switches to the dual simplex method and will quickly reach primal feasibility at the dual optimum point, normally in a small number of iterations.

Warm-starting an interior point method is considerably more difficult, but progress is being made. See Yildirim and Wright (2002) and John and Yildirim (2006) for details.

2.7 Seeking Feasibility and Optimality Simultaneously

An option often provided in simplex-based LP solvers is the ability to seek feasibility and optimality simultaneously. This is what happens when using the big-*M* feasibility-seeking algorithm, of course, but there are better ways to combine the two that avoid the numerical difficulties associated with big-*M*.

The simplest approach is to use a *composite objective* that weights the objective function and a measure of infeasibility, normally the sum of the infeasibilities. The MINOS software (Murtagh and Saunders 1987) uses a composite objective of the form *minimize* $\sigma w(\boldsymbol{cx})$ + (sum of infeasibilities), where $\sigma = 1$ for a minimization objective function and $\sigma = -1$ for a maximization objective function and w is a user-specified weight. If the LP solver reaches an optimum solution for that objective function while the original model remains infeasible, then w is reduced by a factor of 10, and up to five such reductions are allowed before the algorithm gives up.

Infeasible-path interior point algorithms for linear programming have been the subject of a great deal of research in the past decade. Also known as *primal-dual* interior point methods, these algorithms maintain an interior point that satisfies all of the inequality constraints, but that do not necessarily satisfy all of the equality constraints at any point before the optimum is reached. Details are beyond our scope here, but see e.g. Andersen et al. (1996) or Wright (1997).

2.8 Projection Methods

There is a rich and extensive literature on *projection methods* for finding feasible points for sets of constraints that form a convex set, of which sets of linear constraints are an important special case. The properties of these methods are well-studied, including guarantees of convergence for sets of convex inequalities. An excellent reference on this class of methods is Censor and Zenios (1997). Projection methods, under the name of *constraint consensus* methods, are also used as a heuristic technique for reaching near-feasible points in general sets of nonlinear constraints for which the convexity properties are not known (see Sec. 5.5); convergence cannot be guaranteed under these conditions, but the algorithms are remarkably effective.

All methods in this category employ some form of a projection for each violated constraint, most commonly a projection in the gradient or anti-gradient direction. The main idea is to use the gradient of the violated constraint at the current infeasible point, easily given by $\boldsymbol{a}_i$, the ith row of the constraint matrix $\boldsymbol{A}$ in the set of linear constraints $\boldsymbol{Ax}\{\leq, \geq, =\}\boldsymbol{b}$, to calculate the closest point that satisfies the constraint. This closest feasible point is called the *orthogonal projection* of the violated point, and is obtained by moving in the gradient or anti-gradient direction, as appropriate, to the limiting value of the violated constraint (see Sec. 1.2). The vector showing how to move from the current infeasible point to the orthogonal projection point onto an individual violated constraint is sometimes called the *feasibility vector* (Chinneck 2004), and denoted by $\boldsymbol{fv}_i$ for the ith constraint c_i. As has been shown by Xiao et al. (2003) and others, $\boldsymbol{fv}_i = v_i d_i \nabla c_i(\boldsymbol{x}) / \|\nabla c_i(\boldsymbol{x})\|^2$ where:

- $\nabla c_i(\boldsymbol{x})$ is the gradient of the constraint, and $\|\nabla c_i(\boldsymbol{x})\|$ is its length.
- v_i is the *constraint violation* $|c_i(\boldsymbol{x}) - b_i|$, or zero for satisfied constraints,
- d_i is +1 if it is necessary to increase $c(\boldsymbol{x})$ to satisfy the constraint, and –1 if it is necessary to decrease $c_i(\boldsymbol{x})$ to satisfy the constraint.

The squared term in the denominator seems unexpected, but is easily explained. $d_i \nabla c_i(\boldsymbol{x}) / \|\nabla c_i(\boldsymbol{x})\|$ is a unit vector in the gradient or anti-gradient direction, as necessary to reach feasibility. $V_i / \|\nabla c_i(\boldsymbol{x})\|$ is the number of units to move in the appropriate gradient or anti-gradient direction to reach feasibility; the product is $v_i d_i \nabla c_i(\boldsymbol{x}) / \|\nabla c_i(\boldsymbol{x})\|^2$. The length of the feasibility vector for the ith constraint is denoted by $\|\boldsymbol{fv}_i\|$.

The feasibility vectors for the violated constraints are used in different ways in the numerous varieties of projection algorithms (Censor and Zenios 1997; Censor, Elfving and Herman 2001). In all variants, the feasibility vectors must be combined in some way to arrive at an update vector; this final vector is sometimes called the *consensus vector* (Chinneck 2004). Some main algorithm variants are:

- *Sequential* projection algorithms update the current point by finding and applying the feasibility vector for one violated constraint at each iteration. The process continues until feasibility is achieved. The simplest version is cyclic (see below), but other variants are possible, see control sequences below.

- *Simultaneous* projection algorithms calculate the feasibility vector for every violated constraint and then combine them using some form of weighting to determine a final update consensus vector. This process if repeated until feasibility is achieved.
  - In the usual simultaneous projection algorithm, the complete set of feasibility vectors for the violated constraints is combined in a weighted average. *Component averaging* (Censor, Gordon and Gordon 2001) on the other hand, realizes that not all of the constraints contain all of the variables. The final movement vector is therefore computed component-wise, and only the constraints which contain a particular variable are considered when the movement in that dimension is calculated.
- *Control sequences* may be used to adjust which constraints are assessed at each iteration. In a *cyclic* control sequence, a sequential algorithm assesses the constraints in a round-robin fashion. The control sequence may also be *almost cyclic* (constraints or sets of constraints appear in every cycle, but not necessarily in the same order) or *repetitive* (Censor and Zenios 1997). Control sequences may be applied to individual constraints or to sets of constraints.
  - The *most violated constraint control* determines which constraint is currently most violated and uses that constraint in a sequential update algorithm. A similar idea applies in the case of the *remotest set control* which determines a set of constraints that is most violated and uses those constraints in a simultaneous projection algorithm.
  - *Voting heuristics* may be used to determine which subsets of constraints to combine in a simultaneous algorithm (Ibrahim and Chinneck 2005). For example, if the feasibility vectors of more constraints have positive values for some component x_j than negative values, then increase the x_j component by the average value of only the positive x_j components in the feasibility vectors. Several variants of voting methods are described in Sec. 5.5.
- *Relaxation parameters* may be used adjust the length of the consensus vector, either lengthening or shortening it.
- *Oblique projections* may be used instead of orthogonal projections.

A simple example showing several steps in a cyclic orthogonal projectionl projection algorithm for three equality constraints is shown in Fig. 2.1. Fig. 2.2 shows the consensus vector resulting from the component-wise combination of the three orthogonal feasibility vectors.

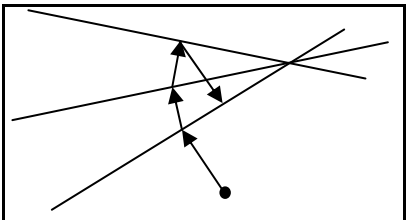

Fig 2.1. Several steps in a cyclic orthogonal projection method

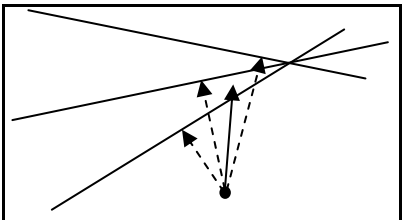

Fig. 2.2. The consensus vector (solid) results from the component-wise averaging of the three feasibility vectors (dashed)

Versions of these algorithms have been introduced by many authors. One of the earlier methods for linear equalities is by Kaczmarz (1937), a cyclic orthogonal projection method. Cimmino (1938) first suggested a fully simultaneous method for linear constraints. Another influential early development was the *relaxation method* for linear inequalities due to Agmon (1954), Motzkin and Schoenberg (1954), which consisted of a cyclic orthogonal projection method with relaxation. See Censor and Zenios (1997) for complete coverage of all related methods.

While projection methods could potentially be used in a feasibility-seeking phase 1 procedure for general linear programs, they have not been adopted for this purpose in commercial LP solvers (though a successive overrelaxation procedure is optionally used as part of a phase 1 procedure in at least one solver: see Sec. 2.5). Instead they have been applied in special-purpose feasibility seeking applications in radiation therapy planning, image reconstruction, etc. many of which are convex nonlinear problems if not linear.

A variant of projection methods known as *randomized thermal relaxation algorithms* is used in the context of finding a maximum cardinality feasible subset for an infeasible set of linear constraints (see Sec. 7.6 for details on the algorithm). Experiments with feasible models comprised of large numbers of linear inequalities show that the method is capable of reaching feasibility or near-feasibility very quickly (Amaldi et al. 2005).