# Practical Guidelines for Solving Difficult Linear Programs

Ed Klotz† • Alexandra M. Newman‡
*†IBM, 926 Incline Way, Suite 100, Incline Village, NV 89451*
*‡Division of Economics and Business, Colorado School of Mines, Golden, CO 80401*
*klotz@us.ibm.com • newman@mines.edu*

**Abstract**

The advances in state-of-the-art hardware and software have enabled the inexpensive, efficient solution of many large-scale linear programs previously considered intractable. However, a significant number of large linear programs can require hours, or even days, of run time and are not guaranteed to yield an optimal (or near-optimal) solution. In this paper, we present suggestions for diagnosing and removing performance problems in state-of-the-art linear programming solvers, and guidelines for careful model formulation, both of which can vastly improve performance.

**Keywords**: linear programming, algorithm selection, algorithmic tuning, numerical stability, memory use, run time, tutorials

## 1 Introduction

Operations research practitioners have been formulating and solving linear programs since the 1940s (Dantzig, 1963). State-of-the-art optimizers such as CPLEX (IBM, 2012), Gurobi (Gurobi, 2012), MOPS (MOPS, 2012), Mosek (MOSEK, 2012), and Xpress-MP (FICO, 2012) can solve most practical large-scale linear programs effectively. However, some "real-world" problem instances require days or weeks of solution time. Furthermore, improvements in computing power and linear programming solvers have often prompted practitioners to create larger, more difficult linear programs that provide a more accurate representation of the systems they model. Although not a guarantee of tractability, careful model formulation and standard linear programming algorithmic tuning often result in significantly faster solution times, in some cases admitting a feasible or near-optimal solution which could otherwise elude the practitioner.

In this paper, we briefly introduce linear programs and their corresponding commonly used algorithms, show how to assess optimizer performance on such problems through the respective algorithmic output, and demonstrate methods for improving that performance through careful formulation and algorithmic parameter tuning. We assume basic familiarity with fundamental

mathematics, such as matrix algebra, and with optimization. We expect that the reader has formulated linear programs and has a conceptual understanding of how the corresponding problems can be solved. The interested reader can refer to basic texts such as Chvátal (1983), Dantzig and Thapa (1997), Rardin (1998), Winston (2004), and Bazaraa et al. (2005) for more detailed introductions to mathematical programming, including geometric interpretations. For a more general discussion of good optimization modeling practices, we refer the reader to Brown and Rosenthal (2008). However, we contend that this paper is self-contained such that relatively inexperienced practitioners can use its advice effectively without referring to other sources.

The remainder of the paper is organized as follows. In Section 2, we introduce linear programs and the simplex and interior point algorithms. We also contrast the performance of these algorithms. In Section 3, we address potential difficulties when solving a linear program, including identifying performance issues from the corresponding algorithmic output, and provide suggestions to avoid these difficulties. Section 4 concludes the paper with a summary. Sections 2.1 and 2.2, with the exception of the tables, may be omitted without loss of continuity for the practitioner interested only in formulation and algorithmic parameter tuning without detailed descriptions of the algorithms themselves. To illustrate the concepts we present in this paper, we show output logs resulting from having run a state-of-the-art optimizer on a standard desktop machine. Unless otherwise noted, this optimizer is CPLEX 12.2.0.2, and the machine possesses four 3.0 gigahertz Xeon chips and eight gigabytes of memory.

## 2 Linear Programming Fundamentals

We consider the following system where $x$ is an $n \times 1$ column vector of continuous-valued, nonnegative decision variables, $A$ is an $m \times n$ matrix of left-hand-side constraint coefficients, $c$ is an $n \times 1$ column vector of objective function coefficients, and $b$ is an $m \times 1$ column vector of right-hand-side data values for each constraint.

$$(P_{LP}): \quad \min \quad c^T x$$

$$\text{subject to} \quad Ax = b$$

$$x \geq 0$$

Though other formats exist, without loss of generality, any linear program can be written in the primal *standard form* we adopt above. Specifically, a maximization function can be changed to a minimization function by negating the objective function (and then negating the resulting

optimal objective). A less-than-or-equal-to or greater-than-or-equal-to constraint can be converted to an equality by adding a nonnegative *slack variable* or subtracting a nonnegative *excess variable*, respectively. Variables that are unrestricted in sign can be converted to nonnegative variables by replacing each with the difference of two nonnegative variables.

We also consider the related *dual problem* corresponding to our primal problem in standard form, $(P_{LP})$. Let $y$ be an $m \times 1$ column vector of continuous-valued, unrestricted-in-sign decision variables; $A$, $b$ and $c$ are data values with the dimensionality given above in $(P_{LP})$.

$$(D_{LP}): \quad \max \quad y^T b$$

$$\text{subject to} \quad y^T A \leq c^T$$

The size of a linear program is given by the number of constraints, $m$, the number of variables, $n$, and the number of non-zero elements in the $A$ matrix. While a large number of variables affects the speed with which a linear program is solved, the commonly used LP algorithms solve linear systems of equations dimensioned by the number of constraints. Because these linear solves often dominate the time per iteration, the number of constraints is a more significant measure of solution time than the number of variables. Models corresponding to practical applications typically contain sparse $A$ matrices in which more than 99% of the entries are zero. In the context of linear programming, a dense matrix need not have a majority of its entries assume non-zero values. Instead, a dense LP matrix merely has a sufficient number or pattern of non-zeros so that the algorithmic computations using sparse matrix technology can be sufficiently time consuming to create performance problems. Thus, a matrix can still have fewer than 1% of its values be non-zero, yet be considered dense. Practical examples of such matrices include those that average more than 10 non-zeros per column and those with a small subset of columns with hundreds of non-zeros. State-of-the-art optimizers capitalize on matrix sparsity by storing only the non-zero matrix coefficients. However, even for such matrices, the positions of the non-zero entries and, therefore, the ease with which certain algorithmic computations (discussed below) are executed, can dramatically affect solution time.

A *basis* for the primal system consists of $m$ variables whose associated matrix columns are linearly independent. The basic variable values are obtained by solving the system $Ax = b$ given the resulting $n - m$ non-basic variables are set to values of zero. The set of the actual values of the variables in the basis, as well as those set equal to zero, is referred to as a *basic solution.* Each primal basis uniquely defines a basis for the dual (see Dantzig (1963), pp. 241-242). For each basis, there is a range of right-hand-side values such that the basis retains the same variables. Within this range, each constraint has a corresponding dual variable value which indicates the change in

the objective function value per unit change in the corresponding right hand side. (This variable value can be obtained indirectly via the algorithm and is readily available through any standard optimizer.) Solving the primal system, ($P_{LP}$), provides not only the primal variable values but also the dual values at optimality; these values correspond to the optimal variable values to the problem ($D_{LP}$). Correspondingly, solving the dual problem to optimality provides both the optimal dual and primal variable values. In other words, we can obtain the same information by solving either ($P_{LP}$) or ($D_{LP}$).

Each vertex (or extreme point) of the polyhedron formed by the constraint set $Ax = b$ corresponds to a *basic solution*. If the solution also satisfies the nonnegativity requirements on all the variables, it is said to be a *basic feasible solution.* Each such basic feasible solution, of which there is a finite number, is a candidate for *an* optimal solution. In the case of multiple optima, any convex combination of extreme-point optimal solutions is also optimal. Because basic solutions contain significantly more zero-valued variables than solutions that are not basic, practitioners may be more easily able to implement basic solutions. On the other hand, basic solutions lack the "diversity" in non-zero values that solutions that are not basic provide. In linear programs with multiple optima, solutions that are not basic may be more appealing in applications in which it is desirable to spread out the non-zero values among many variables. Linear programming algorithms can operate with a view to seeking basic feasible solutions for either the primal or for the dual system, or by examining solutions that are not basic.

## 2.1 Simplex Methods

The practitioner familiar with linear programming algorithms may wish to omit this and the following subsection. The primal simplex method (Dantzig, 1963), whose mathematical details we provide later in this section, exploits the linearity of the objective and convexity of the feasible region in ($P_{LP}$) to efficiently move along a sequence of extreme points until an optimal extreme-point solution is found. The method is generally implemented in two phases. In the first phase, an augmented system is initialized with an easily identifiable extreme-point solution using artificial variables to measure infeasibilities, and then optimized using the simplex algorithm with a view to obtaining an extreme-point solution to the augmented system that is feasible for the original system. If a solution without artificial variables cannot be found, the original linear program is infeasible. Otherwise, the second phase of the method uses the original problem formulation (without artificial variables) and the feasible extreme-point solution from the first phase and moves from that solution to a neighboring, or *adjacent*, solution. With each successive move to another extreme point, the objective function value improves (assuming non-degeneracy, discussed in §3.2) until either: (i) the

algorithm discovers a ray along which it can move infinitely far (to improve the objective) while still remaining feasible, in which case the problem is unbounded, or (ii) the algorithm discovers an extreme-point solution with an objective function value at least as good as that of any adjacent extreme-point solution, in which case that extreme point can be declared *an* optimal solution.

An optimal basis is both primal and dual feasible. In other words, the primal variable values calculated from the basis satisfy the constraints and nonnegativity requirements of ($P_{LP}$), while the dual variable values derived from the basis satisfy the constraints of ($D_{LP}$). The primal simplex method works by constructing a primal basic feasible solution, then working to remove the dual infeasibilities. The dual simplex method (Lemke, 1954) works implicitly on the dual problem ($D_{LP}$) while operating on the constraints associated with the primal problem ($P_{LP}$). It does so by constructing a dual basic feasible solution, and then working to remove the primal infeasibilities. In that sense, the two algorithms are symmetric. By contrast, one can also explicitly solve the dual problem ($D_{LP}$) by operating on the dual constraint set with either the primal or dual simplex method. In all cases, the algorithm moves from one adjacent extreme point to another to improve the objective function value (assuming non-degeneracy) at each iteration.

**Primal simplex algorithm:**

We give the steps of the revised simplex algorithm, which assumes that we have obtained a basis, $B$, and a corresponding initial basic feasible solution, $x_B$, to our system as given in ($P_{LP}$). Note that the primal simplex method consists of an application of the algorithm to obtain such a feasible basis (phase I), and a subsequent application of the simplex algorithm with the feasible basis (phase II).

We define $c_B$ and $A_B$ as, respectively, the vector of objective coefficients and matrix coefficients associated with the basic variables, ordered as the variables appear in the basis. The nonbasic variables belong to the set $N$, and are given by $\{1, 2, ...n\} - \{B\}$.

The revised simplex algorithm mitigates the computational expense and storage requirements associated with maintaining an entire simplex tableau, i.e., a matrix of $A$, $b$, and $c$ components of a linear program, equivalent to the original but relative to a given basis, by computing only essential tableau elements. By examining the steps of the algorithm in the following list, the practitioner can often identify the aspects of the model that dominate the revised simplex method computations, and thus take suitable remedial action to reduce the run time.

1. **Backsolve** Obtain the dual variables by solving the linear system $y^T A_B = c_B^T$, where $A_B$ is represented by an LU factorization. The LU factorization is a product of a lower triangular and upper triangular matrix that is computed through a sequence of pivoting operations

analogous to the operations used to compute the inverse of a matrix. Most simplex algorithm implementations compute and maintain an LU factorization rather than a basis inverse because the former is sparser and can be computed in a more numerically stable manner. See Duff et al. (1986) for details. Given a factorization $A_B = LU$, it follows that $A_B^{-1} = U^{-1}L^{-1}$, so the backsolve is equivalent to solving

$$y^T = c_B^T {A_B}^{-1} = c_B^T U^{-1} L^{-1} \tag{1}$$

This can be calculated efficiently by solving the following sparse triangular linear systems (first, by computing the solution of a linear system involving $U$, and then using the result to solve a linear system involving $L$); this computation can be performed faster than computing $y^T$ as a matrix product of $c^T$ and ${A_B}^{-1}$.

$$p^T U = c_B^T$$
$$y^T L = p^T.$$

2. **Pricing** Calculate the reduced costs $\bar{c}_N^T = c_N^T - y^T A_N$, which indicate for each nonbasic variable the rate of change in the objective with a unit increase in the corresponding variable value from zero.

3. **Entering variable selection** Pick the entering variable $x_t$ and associated incoming column $A_t$ from the set of nonbasic variable indices $N$ with $\bar{c}_N^T < 0$. If $\bar{c}_N^T \geq 0$, stop with an optimal solution $x = (x_B, 0)$.

4. **Forward solve** Calculate the corresponding incoming column $w$ relative to the current basis matrix by solving $A_B w = A_t$.

5. **Ratio test** Determine the amount by which the value of the entering variable can increase from zero without compromising feasibility of the solution, i.e., without forcing the other basic variables to assume negative values. This, in turn, determines the position $r$ of the outgoing variable in the basis and the associated index on the chosen variable $x_j$, $j \in \{1, 2, ...n\}$. Call the outgoing variable in the $r^{th}$ position $x_{j_r}$. Then, such a variable is chosen as follows: $r = \underset{i:w_i>0}{\operatorname{argmin}} \frac{x_{j_i}}{w_i}$.
Let $\theta = \min_{i:w_i>0} \frac{x_{j_i}}{w_i}$.

If there exists no $i$ such that $w_i > 0$, then $\theta$ is infinite; this implies that regardless of the size of the objective function value given by a feasible solution, another feasible solution with a better objective function value always exists. The extreme-point solution given by $(x_B, 0)$, and the direction of unboundedness given by the sum of $(-w, 0)$ and the unit vector $e_t$ combine to form the direction of unboundedness. Hence, stop because the linear program is unbounded. Otherwise, proceed to Step **6**.

6. **Basis update** Update the basis matrix $A_B$ and the associated $LU$ factorization, replacing the outgoing variable $x_{j_r}$ with the incoming variable $x_t$. Periodically refactorize the basis matrix $A_B$ using the factorization given above in Step **1** in order to limit the round-off error (see Section 3.1) that accumulates in the representation of the basis as well as to reduce the memory and run time required to process the accumulated updates.

7. **Recalculate basic variable values** Either update or refactorize. Most optimizers perform between 100 and 1000 updates between each refactorization.

   (a) **Update:** Let $x_t = \theta$; $x_i \leftarrow x_i - \theta \cdot w_i$ for $i \in \{B\} - \{t\}$

   (b) **Refactorize:** Using the refactorization mentioned in Step **1** above, solve $A_B x_B = b$.

8. Return to Step **1**.

A variety of methods can be used to determine the incoming variable for a basis (see Step **3**) while executing the simplex algorithm. One can inexpensively select the incoming variable using *partial pricing* by considering a subset of nonbasic variables and selecting one of those with negative reduced cost. *Full pricing* considers the selection from all eligible variables. More elaborate variable selection schemes entail additional computation such as normalizing each negative reduced cost such that the selection of the incoming variable is based on a scale-invariant metric (Nazareth, 1987). These more elaborate schemes can diminish the number of iterations needed to reach optimality but can also require more time per iteration, especially if a problem instance contains a large number of variables or if the $A$ matrix is dense, i.e., it is computationally intensive to perform the refactorizations given in the simplex algorithm. Hence, if the decrease in the number of iterations required to solve the instance does not offset the increase in time required per iteration, it is preferable to use a simple pricing scheme. In general, it is worth considering non-default variable selection schemes for problems in which the number of iterations required to solve the instance exceeds three times the number of constraints.

Degeneracy in the simplex algorithm occurs when a basic variable assumes a value of zero as it enters the basis. In other words, the value $\theta$ in the minimum ratio test in Step **5** of the simplex

algorithm is zero. This results in iterations in which the objective retains the same value, rather than improving. Theoretically, the simplex algorithm can cycle, revisiting bases multiple times with no improvement in the objective. While cycling is primarily a theoretical, rather than a practical, issue, highly degenerate LPs can generate long, acyclic sequences of bases that correspond to the same objective, making the problem more difficult to solve using the simplex algorithm.

While space considerations preclude us from giving an analogous treatment of the dual simplex method (Fourer, 1994), it is worth noting that the method is very similar to that of the primal simplex method, only preserving dual feasibility while iterating towards primal feasibility, rather than vice versa. The dual simplex algorithm begins with a set of nonnegative reduced costs; such a set can be obtained easily in the presence of an initial basic, dual-feasible solution or by a method analogous to the Phase I primal simplex method. The primal variable values, $x$, are checked for feasibility, i.e., nonnegativity. If they are nonnegative, the algorithm terminates with an optimal solution; otherwise, a negative variable is chosen to exit the basis. Correspondingly, a minimum ratio test is performed on the quotient of the reduced costs and row associated with the exiting basic variable relative to the current basis (i.e., the simplex tableau row associated with the exiting basic variable). The ratio test either detects infeasibility or identifies the incoming basic variable. Finally, a basis update is performed on the factorization (Bertsimas and Tsitsiklis, 1997).

## 2.2 Interior Point Algorithms

The earliest interior point algorithms were the affine scaling algorithm proposed by Dikin (1967) and the logarithmic barrier algorithm proposed by Fiacco and McCormick (1968). However, at that time, the potential of these algorithms for efficiently solving large-scale linear programs was largely ignored. The ellipsoid algorithm, proposed by Khachian (1979), established the first polynomial time algorithm for linear programming. But, this great theoretical discovery did not translate to good performance on practical problems. It wasn't until Karmarkar's projective method (Karmarkar, 1984) had shown great practical promise and was subsequently demonstrated to be equivalent to the logarithmic barrier algorithm (Gill et al., 1986), that interest in these earlier interior point algorithms increased. Subsequent implementations of various interior point algorithms revealed primal-dual logarithmic barrier algorithms as the preferred variant for solving practical problems (Lustig et al., 1994).

None of these interior point algorithms or any of their variants uses a basis. Rather, the algorithm searches through the interior of the feasible region, avoiding the boundary of the constraint set until it finds an optimal solution. Each variant possesses a different means for determining a search direction. However, all variants fundamentally rely on centering the current iterate, com-

puting an improving search direction, moving along it for a given *step size* short enough that the boundary is not reached (until optimality), and then recentering the iterate.

While not the most efficient in practice, Dikin's Primal Affine Scaling Algorithm provides the simplest illustration of the computational steps of these interior point algorithms. Therefore, we describe Dikin's Algorithm in detail below. Lustig et al. (1990) contains a more detailed description of the more frequently implemented primal-dual logarithmic barrier algorithm.

The $k^{th}$ iteration of Dikin's algorithm operates on the linear program ($P_{LP}$), along with a feasible interior point solution $x_k > 0$. The algorithm centers the feasible interior point by rescaling the variables based on the values of $x_k$, computes a search direction by projecting the steepest descent direction onto the null space of the rescaled constraint matrix, and moves in the search direction while ensuring that the new scaled solution remains a feasible interior point. The algorithm then unscales the solution, resulting in a new iterate, $x_{k+1}$. Following the unscaling, the algorithm performs a convergence test for optimality on $x_{k+1}$. If $x_{k+1}$ is not optimal, the algorithm repeats its steps using $x_{k+1}$ as the feasible interior point solution. The following steps provide the mathematical details.

**Interior point algorithm with affine scaling:**

1. **Centering** Let $D = Diag(x_k)$. Rescale the problem to center the current interior feasible solution by letting $\hat{A} = AD$, $\hat{c}^T = c^T D$. Hence, $\hat{x}_k = D^{-1}x_k = e$, the vector consisting of all 1's. Note that $\hat{A}\hat{x}_k = b$.

2. **Search Direction Computation** For the rescaled problem, project the steepest descent direction $-\hat{c}^T$ onto the null space of the constraint matrix $\hat{A}$, resulting in the search direction $p_k = -(I - \hat{A}^T(\hat{A}\hat{A}^T)^{-1}\hat{A})\hat{c}$.

3. **Step Length** Add a positive multiple $\theta$ of the search direction to $p_k$, the scaled interior feasible point, by computing $\hat{x}_{k+1} = e + \theta p_k$. If $p_k \geq 0$, then $\hat{x}_{k+1}$, and hence $x_{k+1}$, can increase without bound; stop the algorithm with an unbounded solution. Otherwise, because $\hat{A}p_k = 0$, $\hat{A}x_{k+1} = b$. Therefore, $\theta$ must be chosen to ensure that $\hat{x}_{k+1} > 0$. For any constant $\alpha$ such that $0 < \alpha < 1$, the update $\hat{x}_{k+1} = e - (\frac{\alpha}{\min_j p_k[j]})p_k$ suffices.

4. **Optimality Test** Unscale the problem, setting $x_{k+1} = D\hat{x}_{k+1}$. Test $x_{k+1}$ for optimality by checking whether $||x_{k+1} - x_k||$ is suitably small. If $x_{k+1}$ is optimal, stop the algorithm. Otherwise, return to Step **1** with feasible interior point solution $x_{k+1}$.

The calculation of $p_k = -(I - \hat{A}^T(\hat{A}\hat{A}^T)^{-1}\hat{A})\hat{c}$ (Step **2**) is the most time consuming operation of this algorithm. First, one must perform the matrix vector multiplication $v = \hat{A}\hat{c}$. Then, one

must compute the solution $w$ to the linear system of equations $(\hat{A}\hat{A}^T)w = v$. This step typically dominates the computation time of an iteration. Subsequently, one must perform a second matrix vector multiplication, $\hat{A}^T w$, then subtract $\hat{c}$ from the result.

The simplex algorithm creates an $m \times m$ basis matrix of left-hand-side coefficients, $A_B$, which is invertible. By contrast, interior point algorithms do not maintain a basis matrix. The matrix $AA^T$ is not guaranteed to be invertible unless $A$ has full rank. Fortunately, in the case of solving an LP, full rank comes naturally, either through removal of dependent constraints during presolve, or because of the presence of slack and artificial variables in the constraint matrix.

Other interior point algorithms maintain feasibility by different means. Examples include applying a logarithmic barrier function to the objective rather than explicitly projecting the search direction onto the null space of the rescaled problem, and using a projective transformation instead of the affine transformation of Dikin's algorithm to center the iterate. Some also use the primal and dual constraints simultaneously, and use more elaborate methods to calculate the search direction in order to reduce the total number of iterations. However, in all such practical variants to date, the dominant calculation remains the solution of a system of linear equations similar to the form $(\hat{A}\hat{A}^T)w = v$, as in Dikin's algorithm. As of the writing of this paper, the primal dual barrier algorithm, combined with Mehrotra's predictor-corrector method (Mehrotra, 1992), has emerged as the method of choice in most state-of-the-art optimizers.

Because the optimal solutions to linear programs reside on the boundary of the feasible region, interior point algorithms cannot move to the exact optimal solution. They instead rely on convergence criteria. Methods to identify an optimal basic solution from the convergent solution that interior point algorithms provide have been developed (Megiddo, 1991). A procedure termed *crossover* can be invoked in most optimizers to transform a (typically near optimal) interior solution to an optimal extreme-point solution. (In the case of a unique solution, the interior point method would converge towards the optimal extreme-point solution, i.e., a basic solution.) Crossover provides solutions that are easier to implement in practice. While crossover typically comprises a small percentage of the run time on most models, it can be time consuming, particularly when initiated on a suboptimal interior point solution with significant distance from an optimal solution. For primal-dual interior algorithms (Wright, 1997), the optimality criterion is usually based on a normalized duality gap, e.g., the quotient of the duality gap and primal objective (or dual objective since they are equal at optimality): $\frac{(c^T x - y^T b)}{c^T x}$.

In order to solve the linear system $(\hat{A}\hat{A}^T)w = v$ efficiently, most practical implementations maintain a Cholesky factorization $(\hat{A}\hat{A}^T) = \hat{L}\hat{L}^T$. The non-zero structure, i.e., the positions in the matrix in which non-zero elements of $\hat{A}\hat{A}^T$ lie, profoundly influences interior point algorithm run

$$\begin{pmatrix} x & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 \\ x & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & x \end{pmatrix} * \begin{pmatrix} x & x & x & x & x \\ 0 & x & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & x \end{pmatrix} = \begin{pmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{pmatrix}$$

$$\begin{pmatrix} x & x & x & x & x \\ 0 & x & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & x \end{pmatrix} * \begin{pmatrix} x & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 \\ x & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & x \end{pmatrix} = \begin{pmatrix} x & x & x & x & x \\ x & x & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 \\ x & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & x \end{pmatrix}$$

Figure 1: Let $x$ denote an arbitrarily valued non-zero entry in the above matrices. The upper matrix product yields a completely dense result, while the lower matrix product yields a reasonably sparse result. The two products are merely reversals of each other.

times. Since $D$ is a diagonal matrix, $\hat{A}\hat{A}^T$ and $AA^T$ have the same non-zero structure. For sparse matrices, the non-zero structure of $AA^T$ depends heavily on the structure of the original matrix $A$. Note that even if the $A$ matrix is sparse, the product of $A$ and $A^T$ can be dense. Consider the two matrix products in Figure 1, in which $x$ represents an arbitrarily valued non-zero matrix entry. The first product of $A$ and $A^T$ leads to a fully dense matrix (barring unlikely cancellation of terms in some entries), while the second product, which is simply a reversal of the two matrices $A$ and $A^T$, remains relatively sparse.

A significant amount of work involves determining how to order $A$ so that the Cholesky factor of $AA^T$ is as sparse as possible. Although the non-zero values of $\hat{A}$ change from one iteration to the next, the non-zero structure of $AA^T$ remains unchanged throughout the algorithm. So, interior point algorithm implementations can perform a large part of the computation that involves $AA^T$ at the start of the algorithm by computing a symbolic Cholesky factorization of $AA^T$ to provide the static non-zero structure of a lower triangular matrix $L$ such that: $AA^T = LL^T$. This avoids recomputing $(\hat{A}\hat{A}^T)^{-1}$ from scratch during each iteration and removes the need to maintain any inverse at all. Rather, the implementation uses the Cholesky factorization to solve the system of equations $(\hat{A}\hat{A}^T)w = v$. At each iteration, the algorithm must update the individual values of the Cholesky factorization, but the non-zero structure of the factorization is known from the initial symbolic factorization.

## 2.3 Algorithm Performance Contrast

The linear programming algorithms we have discussed perform differently depending on the characteristics of the linear programs on which they are invoked. Although we introduce ($P_{LP}$) in standard form with equality constraints, yielding an $m \times n$ system with $m$ equality constraints and $n$ variables, we assume for the following discussion that our linear program is given as naturally formulated, i.e., with a mixture of equalities and inequalities, and, as such, contains $m$ equality and inequality constraints and $n$ variables.

The solution time for simplex algorithm iterations is more heavily influenced by the number of constraints than by the number of variables. This is because the row-based calculations in the simplex algorithm involving the $m \times m$ basis matrix usually comprise a much larger percentage of the iteration run time than the column-based operations. One can equivalently solve either the primal or the dual problem, so the practitioner should select the one that most likely solves fastest. Some optimizers have the ability to create the dual model before or after presolving a linear program, and to use internal logic for determining when to solve the explicit dual. The aspect ratio, $\frac{n}{m}$, generally indicates whether solving the primal or the dual problem, and/or solving the problem with the primal or the dual simplex method is likely to be more efficient. If $m << n$, it is more expedient to preserve the large aspect ratio. In this case, the primal simplex algorithm with partial pricing is likely to be effective because reduced costs on a potentially small subset of the nonbasic variables need to be computed at each iteration. By contrast, the dual simplex algorithm must compute a row of the simplex tableau during the ratio test that preserves dual feasibility. This calculation in the dual simplex algorithm involves essentially the same effort as full pricing in the primal simplex algorithm. This can consume a large portion of the computation time of each iteration. For models with $m >> n$, an aspect ratio of 0.5 or smaller indicates that solving the dual explicitly yields faster performance than the dual simplex method, though solving the dual explicitly can also be faster even if the aspect ratio is greater than 0.5. Also, under primal degeneracy, implicitly solving the dual problem via the dual simplex method or solving the explicit dual can dramatically reduce the iteration count.

Regarding interior point algorithms, they may be effective when $m << n$. And, when $m$ is relatively small, $AA^T$ has only $m$ rows and $m$ columns, which gives the interior point algorithm a potential advantage over simplex algorithms, even when $AA^T$ is relatively dense. For $m >> n$, the interior point algorithm applied to the dual problem has potential to do very well. If $A$ has either dense rows or dense columns, choosing between solving the primal and the dual affects only the interior point algorithm performance. Neither the primal nor the dual simplex algorithm and the associated factorized basis matrices has an obvious advantage over the other on an LP (or its

dual) with dense rows or columns. However, in the interior point method, if $AA^T$ is dense, and if the Cholesky factor is also dense, explicitly solving the dual model might produce faster run time performance. Dense columns in the primal problem, which frequently result in a dense $AA^T$ matrix, become dense rows in the dual problem, which are less likely to result in a dense $AA^T$ matrix (see Figure 1). An $A$ matrix which averages more than ten non-zero elements per column (regardless of the values of $m$ and $n$), can make problems difficult to solve.

Most state-of-the-art optimizers can handle a modest number of dense columns by representing $AA^T$ as the sum of outer products of the columns of $A$ and separating the sum of the outer products of the dense columns. See Wright (1997) for more details. However, this approach can exhibit numerical instability (see Section 3.1) as the number of dense columns increases, so applying an interior point method to the explicit dual problem may be more effective.

Iteration Log #1 sheds light on the density of $AA^T$, as well as the additional non-zero matrix elements introduced during the factorization. The most important information in this output is the number of non-zeros in the lower triangle of $AA^T$ and the total number of non-zeros in the Cholesky factor. An interior point algorithm is less likely to outperform the simplex algorithm if the number of non-zeros in the lower triangle of $AA^T$ is much larger than the number of non-zeros in $A$, or if the number of non-zeros in the resulting Cholesky factor grows dramatically relative to the number of non-zeros in $AA^T$. Not surprisingly, then, the output in Iteration Log #1 suggests that an interior point algorithm is likely to outperform the simplex algorithm, since both the density of $AA^T$ and the fill-in from the resulting factorization are quite modest. That is indeed true for this model (south31, a publicly available model from `http://www.sztaki.hu/~meszaros/public_ftp/lptestset/misc/`. CPLEX's barrier method solves the model in just under one second, in contrast to 7 and 16 seconds for the primal and dual simplex methods, respectively.

---

**Iteration Log #1**

```
Reduced LP has 17645 rows, 35223 columns, and 93047 non-zeros.
Presolve time =    0.05 sec.
Parallel mode: using up to 4 threads for barrier.
***NOTE: Found 144 dense columns.
Number of non-zeros in lower triangle of A*A' = 31451
Using Approximate Minimum Degree ordering
Total time for automatic ordering = 0.00 sec.
Summary statistics for Cholesky factor:
```

```
  Threads                     = 4
  Rows in Factor              = 17789
  Integer space required      = 70668
  Total non-zeros in factor = 116782
  Total FP ops to factor      = 2587810
```

---

Now consider Iteration Log #2 which represents, in fact, output for the same model instance as considered in Iteration Log #1, but with dense column handling disabled. The $A$ matrix has $93,047$ non-zeros, while the lower triangle of $AA^T$ from which the Cholesky factor is computed has over 153 million non-zeros, which represents a huge increase. The additional fill-in associated with the factorization is relatively negligible, with $153,859,571 - 153,530,245 = 329,326$ additional non-zeros created.

---

**Iteration Log #2**

```
Reduced LP has 17645 rows, 35223 columns, and 93047 non-zeros.
Presolve time =    0.05 sec.
Parallel mode: using up to 4 threads for barrier.
Number of non-zeros in lower triangle of A*A' = 153530245
Using Approximate Minimum Degree ordering
Total time for automatic ordering = 11.88 sec.
Summary statistics for Cholesky factor:
  Threads                     = 4
  Rows in Factor              = 17645
  Integer space required      = 45206
  Total non-zeros in factor = 153859571
  Total FP ops to factor      = 1795684140275
```

---

The barrier algorithm must now perform calculations with a Cholesky factor containing over 153 million non-zeros instead of fewer than $120,000$ non-zeros with the dense column handling. This increases the barrier run time on the model instance from less than one second to over 19 minutes.

Iteration Log #3 illustrates the case in which the non-zero count of $AA^T$ (566,986) is quite modest relative to the non-zero count in $A$; however, the subsequent fill-in while computing the Cholesky factor is significant, as can be seen from the final non-zero count, which exceeds 20 million. And, indeed, the barrier algorithm requires more than 4 times as long as the dual simplex method to solve this model instance.

---

**Iteration Log #3**

```
Reduced LP has 79972 rows, 404517 columns, and 909056 non-zeros.
Presolve time =    1.57 sec.
Parallel mode: using up to 16 threads for barrier.
Number of non-zeros in lower triangle of A*A' = 566986
Using Nested Dissection ordering
Total time for automatic ordering = 9.19 sec.
Summary statistics for Cholesky factor:
  Threads                  = 16
  Rows in Factor           = 79972
  Integer space required   = 941792
  Total non-zeros in factor = 20737911
  Total FP ops to factor   = 24967192039
```

---

Finally, Iteration Log #4 illustrates the potential advantage of an interior point method over the simplex methods as the problem size grows very large. Due to its polynomial behavior, an interior point method's iteration count tends to increase slowly as problem size increases. Iteration Log #4 contains results for the zib05 model, available in MPS format at `http://miplib.zib.de/contrib/miplib2003-contrib/IPWS2008/`. Despite over 9 million columns in $A$ with an average density of over 35 non-zeros per column, the modest number of rows and level of fill-in when computing the Cholesky factor results in a manageable model, provided sufficient memory (about 20 gigabytes) is available. Due to the increased memory needed to solve this model, this run was done on a machine with 4 2.9 gigahertz quad core Xeon chips and 128 gigabytes of memory.

---

**Iteration Log #4**

```
Reduced LP has 9652 rows, 9224120 columns, and 348547188 non-zeros.
Presolve time =  268.99 sec.
Parallel mode: using up to 16 threads for barrier.
Number of non-zeros in lower triangle of A*A' = 16747402
Using Nested Dissection ordering
Total time for automatic ordering = 409.44 sec.
Summary statistics for Cholesky factor:
  Threads                  = 16
  Rows in Factor           = 9652
  Integer space required   = 256438
  Total non-zeros in factor = 36344797
  Total FP ops to factor   = 171936037267
```

---

By contrast, the simplex methods must select a basis of 9652 columns among the 9.2 million available, making this model much more challenging. CPLEX's barrier algorithm can solve this model in about 3 hours, while neither the primal nor the dual simplex method solves the model on the same machine in 20 hours.

The simplex method that can start with a feasible basis has a performance advantage over the one that lacks a feasible basis. Although, at the time of this writing, interior point methods cannot take advantage of a starting solution from a modified problem, the computational steps they perform parallelize better than those in the simplex algorithm. The simplex algorithm possesses a basis factorization whose non-zero structure changes at each iteration, while the interior point algorithm maintains a factorization with static non-zero structure throughout the algorithm that is more amenable to parallelization. The ability to parallelize an algorithm can produce faster run time performance.

Both the implicit dual, i.e., the dual simplex algorithm, and solving the dual explicitly, offer the advantages of (i) better performance in the presence of primal degeneracy, and (ii) a more easily available feasible starting basis on practical models. In many practical cases, dual feasibility may be easier to obtain than primal feasibility. If, as is often the case in practical models involving costs, all values of $c$ are nonnegative, then $y = 0$ is immediately dual feasible. In fact, this holds even if the dual constraints are a mixture of equalities and inequalities (in either direction).

The basis with which the simplex algorithm operates allows for "warm starts," i.e., the ability to take advantage of a solution as a starting point for a slightly modified problem. Such a problem can

be found as a subproblem in an iterative technique in which new variables and/or constraints have been inserted. For the case in which the practitioner is interested in multiple, similar solves, the choice of the primal or the dual simplex method is influenced by how the modifications that generate the sequence of instances preserve primal or dual feasibility of the basis and associated solution from the previous solve. For example, adding columns to ($P_{LP}$) maintains primal feasibility, but typically compromises dual feasibility. One can set the variables associated with the newly added columns to nonbasic at zero and append those values to the values of the existing primal feasible solution. Therefore, after adding columns, the primal simplex method has an advantage over the dual simplex method because the optimal basis from the previous LP remains primal feasible, but not typically dual feasible. Dual feasibility would also be preserved if the additional columns corresponded to redundant constraints in the dual, but such additions are uncommon in practice. Analogously, adding rows to a dual feasible solution (which is equivalent to adding columns to the explicit dual LP) preserves dual feasibility but typically compromises primal feasibility, so the dual simplex method has an advantage over the primal simplex method. Similarly, one can examine the different types of problem modifications to an LP and determine whether the modification preserves primal or dual feasibility. Table 1 summarizes the different types of modifications to the primal problem, ($P_{LP}$), and whether they ensure preservation of primal or dual feasibility. Note that the preservation of primal feasibility for a row of the table does not imply the absence of dual feasibility for the given problem modification (or vice versa). For some problem modifications, additional conclusions about preservation of feasibility can be made with a closer examination of the nature of the change. For example, changing an objective coefficient that relaxes a dual constraint preserves dual feasibility when the associated variable is nonbasic, but can result in a dual infeasible basis if said variable is basic.

| Problem modification | Primal or dual feasibility preserved? |
|---|---|
| Add columns | Primal feasibility |
| Add rows | Dual feasibility |
| Change objective function coefficients | Primal feasibility |
| Change right-hand-side coefficients | Dual feasibility |
| Change matrix coefficients of basic variables | Neither |
| Change matrix coefficients of nonbasic variables | Primal feasibility |

Table 1: Different types of modifications to ($P_{LP}$) influence primal or dual feasibility of the modified LP.

Table 2 summarizes guidelines for the circumstances under which one should use primal simplex, dual simplex, or interior point method; the characteristics we present in this table are not mutually exclusive. For example, an LP may have $m << n$ but also have dense columns in the $A$ matrix.

Table 2 recommends either the primal simplex method on the primal problem, or an interior point method on the primal or dual problem. However, the dense columns potentially hinder the interior point method on the primal problem, while the large aspect ratio is potentially problematic for the dual problem. By contrast, there are no obvious limitations to primal simplex method performance. Therefore, while one should consider all of these recommendations, the primal simplex method on the primal LP has the most promise in this example.

| Characteristic | Recommended method |
|---|---|
| $m << n$ | Primal simplex or interior point on primal problem |
| $m >> n$ | Primal simplex or interior point on dual problem |
| Dense rows in $A$ matrix | Solve primal problem if using interior point |
| Dense columns in $A$ matrix | Solve dual problem if using interior point |
| Availability of parallel hardware | Interior point |
| Multiple solves on similar instances necessary | Primal or dual simplex |
| Availability of a primal or dual feasible basis | Primal or dual simplex, respectively |
| Minimization with nonnegative $c$ | Dual simplex as dual feasible basis is available |

Table 2: Under various circumstances, different methods have a greater chance of faster solution time on a linear programming problem instance.

# 3 Guidelines for Successful Algorithm Performance

## 3.1 Numerical Stability and Ill Conditioning

Because most commonly used computers implement floating point computations in finite precision, arithmetic calculations such as those involved in solving linear programming problems can be prone to inaccuracies due to round-off error. Round-off error can arise from numerical instability or ill conditioning. In general terms, ill conditioning pertains to the situation in which a small change to the input can result in a much larger change to the output in models or systems of equations (linear or otherwise). Ill conditioning can occur under perfect arithmetic as well as under finite precision computing. Numerical stability (or lack thereof) is a characteristic of procedures and algorithms implemented under finite precision. A procedure is numerically stable if its backward error analysis results in small, bounded errors on all data instances, i.e., if a small, bounded perturbation to the model would make the computed solution to the unperturbed model an exact solution. Thus, numerical instability does not imply ill conditioning, nor does ill conditioning imply numerical instability. But, a numerically unstable algorithm introduces larger perturbations into its calculations than its numerically stable counterpart; this can lead to larger errors in the final computed solution if the model is ill conditioned. See Higham (1996) for more information on the different types of error analysis and their relationships to ill conditioning.

The practitioner cannot always control the floating point implementations of the computers on which he works and, hence, how arithmetic computations are done. As of this writing, exact floating point calculations can be done, but these are typically done in software packages such as Maple (Cybernet Systems Co., 2012) and Mathematica (Wolfram, 2012), which are not large-scale linear programming solvers. The QSopt optimizer (Applegate et al., 2005) reflects significant progress in exact linear programming, but even this solver still typically performs some calculations in finite precision. Regardless, the practitioner can and should be aware of input data and the implications of using an optimization algorithm and a floating point implementation on a model instance with such data. To this end, let us consider the derivation of the condition number of a square matrix, and how ill conditioning can affect the optimization of linear programs on finite-precision computers.

Consider a system of linear equations in standard form, $A_B x_B + A_N x_N = b$, where $B$ constitutes the set of basic variables, $N$ constitutes the set of non-basic variables, and $A_B$ and $A_N$ are the corresponding left-hand-side basic and non-basic matrix columns, respectively. Equivalently, $x_B$ and $x_N$ represent the vectors of basic and non-basic decision variables, respectively. We are

interested in solving ($P_{LP}$), whose constraints we can rewrite as follows:

$$A_B x_B = b - A_N x_N = b \tag{2}$$

Note that in ($P_{LP}$), all variables have lower bounds of zero and infinite upper bounds. Therefore, all nonbasic variables are zero and $A_N x_N = 0$. By contrast, if some of the variables have finite non-zero lower and/or upper bounds, then variables at these bounds can also be nonbasic and not equal to zero. Also note that equation (2) corresponds to Step **7b** of the previously provided description of the primal simplex algorithm. In addition, Steps **1** and **4** solve similar systems of linear equations involving the basis matrix, $A_B$.

The exact solution of equation (2) is given by:

$$x_B = A_B^{-1} b \tag{3}$$

Consider a small perturbation, $\Delta b$, to the right hand side of equations (2). We wish to assess the relation between $\Delta b$ and the corresponding change $\Delta x_B$ to the computed solution of the perturbed system of equations:

$$A_B(x_B + \Delta x_B) = b + \Delta b \tag{4}$$

The exact solution of this system of equations (4) is given by:

$$(x_B + \Delta x_B) = A_B^{-1}(b + \Delta b) \tag{5}$$

Subtracting equations (3) from those given in (5), we obtain:

$$\Delta x_B = A_B^{-1} \Delta b \tag{6}$$

Applying the Cauchy-Schwarz inequality to equations (6), we obtain:

$$||\Delta x_B|| \leq ||A_B^{-1}||||\Delta b|| \tag{7}$$

In other words, the expression (7) gives an upper bound on the maximum absolute change in $x_B$ relative to that of $b$. Similarly, we can get a relative change in $x_B$ by applying the Cauchy-Schwarz inequality to equation (2):

$$||b|| \leq ||A_B||||x_B|| \tag{8}$$

Multiplying the left and right hand sides of (7) and (8) together, and rearranging terms:

$$\frac{||\Delta x_B||}{||x_B||} \leq ||A_B||||A_B^{-1}||\left(\frac{||\Delta b||}{||b||}\right) \tag{9}$$

From (9), we see that the quantity $\kappa = ||A_B||||A_B^{-1}||$ is a scaling factor for the relative change in the solution, $\frac{||\Delta x_B||}{||x_B||}$, given a relative change in the right hand side, $\frac{||\Delta b||}{||b||}$. The quantity $\kappa\left(\frac{||\Delta b||}{||b||}\right)$ provides an upper bound on the relative change in the computed solution, $\frac{||\Delta x_B||}{||x_B||}$, for a given relative change in the right hand side, $\left(\frac{||\Delta b||}{||b||}\right)$. Recall that ill conditioning in its most general sense occurs when a small change in the input of a system leads to a large change in the output. The quantity $\kappa$ defines the *condition number* of the matrix $A_B$ and enables us to assess the ill conditioning associated with the system of equations in (2). Larger values of $\kappa$ imply greater potential for ill conditioning in the associated square linear system of equations by indicating a larger potential change in the solution given a change in the (right-hand-side) inputs. Because both the simplex and interior point algorithms need to solve square systems of equations, the value of $\kappa$ can help predict how ill conditioning affects the computed solution of a linear program. Note that the condition number $\kappa$ has the same interpretation when applied to small perturbations in $A_B$.

In practice, perturbations $\Delta b$ can occur due to (i) finite precision in the computer on which the problem is solved, (ii) round-off error in the calculation of the input data to the problem, or (iii) round-off error in the implementation of the algorithm used to solve the problem. Note that these three issues are related. Input data with large differences in magnitude, even if computed precisely, require more shifting of exponents in the finite precision computing design of most computers than data with small differences in magnitude. This typically results in more round-off error in computations involving numbers of dramatically different orders of magnitude. The increased round-off error in these floating point calculations can then increase the round-off error that accumulates in the algorithm implementation. Most simplex algorithm implementations are designed to reduce the level of such round-off error, particularly as it occurs in the ratio test and LU factorization calculations. However, for some LPs, the round-off error remains a problem, regardless of the efforts to contain it in the implementation. In this case, additional steps must be taken to reduce the error by providing more accurate input data, by improving the conditioning of the model, or by tuning algorithm parameters. While this discussion is in the context of a basis matrix of the simplex algorithm, the calculations in equations (3)-(9) apply to any square matrix and therefore also apply to the system of equations involving $AA^T$ in the barrier algorithm.

Let us consider the condition number of the optimal basis to a linear program. Given the

typical machine precision of $10^{-16}$ for double precision calculations and equation (9) that defines the condition number, a condition number value of $10^{10}$ provides an important threshold value. Most state-of-the-art optimizers use default feasibility and optimality tolerances of $10^{-6}$. In other words, a solution is declared feasible when solution values that violate the lower bounds of 0 in $(P_{LP})$ do so by less than the feasibility tolerance. Similarly, a solution is declared optimal when any negative reduced costs in $(P_{LP})$ are less (in absolute terms) than the optimality tolerance. Because of the values of these tolerances, condition numbers of $10^{10}$ or greater imply a level of ill conditioning that could cause the implementation of the algorithm to make decisions based on round-off error. Because (9) is an inequality, a condition number of $10^{10}$ does not guarantee ill conditioning, but it provides guidance as to when ill conditioning is likely to occur.

Round-off error associated with finite precision implementations depends on the order of magnitude of the numbers involved in the calculations. Double precision calculations involving numbers with orders of magnitude larger than $10^0$ can introduce round-off error that is larger than the machine precision. However, because machine precision defines the smallest value that distinguishes two numbers, calculations involving numbers with smaller orders of magnitude than $10^0$ still possess round-off errors at the machine precision level. For example, for floating point calculations involving at least one number on the order of $10^5$, round-off error due to machine precision can be on the order of $10^5 * 10^{-16} = 10^{-11}$. Thus, round-off error for double precision calculations is relative, while most optimizers use absolute tolerances for assessing feasibility and optimality. State-of-the-art optimizers typically scale the linear programs they receive to try to keep the round-off error associated with double precision calculations close to the machine precision. Nonetheless, the practitioner can benefit from formulating LPs that are well scaled, avoiding mixtures of large and small coefficients that can introduce round-off errors significantly larger than machine precision. If this is not possible, the practitioner may need to consider solving the model with larger feasibility or optimality tolerances than the aforementioned defaults of $10^{-6}$.

Equations $(3) - (9)$ above are all done under perfect arithmetic. Finite-precision arithmetic frequently introduces perturbations to data. If a perturbation to the data is on the order of $10^{-16}$ and the condition number is on the order of $10^{12}$, then round-off error as large as $10^{-16} * 10^{12} = 10^{-4}$ can creep into the calculations, and linear programming algorithms may have difficulty distinguishing numbers accurately within its $10^{-6}$ default optimality tolerance. The following linear program provides an example of ill conditioning and round-off error in the input data:

$$\max x_1 + x_2 \tag{10}$$

$$\text{subject to } \frac{1}{3}x_1 + \frac{2}{3}x_2 = 1 \tag{11}$$

$$x_1 + 2x_2 = 3 \tag{12}$$

$$x_1, x_2 \geq 0 \tag{13}$$

Because equations (11) and (12) are linearly dependent, $\{x_1,\ x_2\}$ cannot form a basis. Rather, $x_1$ is part of the optimal basis, along with an artificial variable, and the optimal solution under perfect precision in this case is $x_1^* = 3$ and $x_2^* = 0$. By contrast, imprecisely rounded input with coefficients on $x_1$ and $x_2$ of 0.333333 and 0.666667, respectively, in the first constraint produces no linear dependency in the constraint sets (11) and (12), allowing both $x_1$ and $x_2$ to be part of a feasible basis. Furthermore, the optimal solution under perfect precision, $x_1^* = 3$ and $x_2^* = 0$, is infeasible in (11) with the rounded coefficients (because $3 \times 0.333333 \neq 1$). Instead, a solution of $x_1^* = x_2^* = 1$ satisfies both (11) and (12) using the default feasibility tolerance of $10^{-6}$ and is, in fact, optimal in this case. The associated optimal basis is $\{x_1,\ x_2\}$, with condition number $8.0 * 10^6$. Using three more digits of accuracy on the coefficients of $x_1$ and $x_2$ in constraint (11) yields the correct optimal solution. The optimal basis now consists of $x_1$ and one of the artificial variables, resulting in a condition number of 8.0. However, the better approach is to avoid rounding (or approximating) altogether by appropriately scaling the problem (i.e., multiplying through by the denominator) such that the first constraint is expressed as $x_1 + 2x_2 = 3$. When the first constraint is expressed in this way, constraints (11) and (12) are obviously redundant, which simplex algorithm implementations can easily handle. By contrast, the representation using rounding yields a near-singular basis matrix, $A_B$.

When a linear program is poorly scaled or ill conditioned, the simplex algorithm may lose feasibility when solving the problem instance. Even if the instance remains feasible, the algorithm might try to refactorize $A_B = LU$ to regain digits of accuracy it may have lost due to round-off error in the basis updates. Iteration Log #5, involving a model with an objective to be maximized, illustrates an instance that loses feasibility. The log prints the objective function value each time the algorithm refactorizes, rather than updates, $A_B$. The algorithm tries to refactorize the basis matrix three times in three iterations, and increases the Markowitz threshold to improve the accuracy with which these computations are done. Larger values of the Markowitz threshold impose tighter restrictions on the criteria for a numerically stable pivot during the computation of the LU factorization (Duff et al., 1986). However, despite the increased accuracy in the factorization

starting at iteration 6391, enough round-off error accumulates in the subsequent iterations so that the next refactorization, at iteration 6456, results in a loss of feasibility.

---

**Iteration Log #5**

```
Iter: 6389 Objective  = 13137.039899
Iter: 6390 Objective  = 13137.039899
Iter: 6391 Objective  = 13726.011591
Markowitz threshold set to 0.3.
Iter: 6456 Scaled infeas = 300615.030682
...
Iter: 6752 Scaled infeas = 0.000002
Iter: 6754 Objective = -23870.812630
```

---

Although the algorithm regains feasibility at iteration 6754, it spends an extra 298 (6754 - 6456) iterations doing so, and the objective function is much worse than the one at iteration 6389. Additional iterations are then required just regain the previously attained objective value. So, even when numerical instability or ill conditioning does not prevent the optimizer from solving the model to optimality, it may slow down performance significantly. Improvements to the model formulation by reducing the source of perturbations and, if needed, changes to parameter settings can reduce round-off error in the optimizer calculations, resulting in smaller basis condition numbers and faster computation of optimal solutions.

As another caveat, the practitioner should avoid including data with meaningful values smaller than the optimizer's tolerances. Similarly, the practitioner should ensure that the optimizer's tolerances exceed the largest round-off error in any of the data calculations. The computer can only handle a fixed number of digits. Very small numerical values force the algorithm to make decisions about whether those smaller values are real or due to round-off error, and the optimizer's decisions can depend on the coordinate system (i.e., basis) with which it views the model. Consider the following feasibility problem:

$$c_1 : -x_1 + 24x_2 \leq 21 \tag{14}$$

$$-\infty < x_1 \leq 3 \tag{15}$$

$$x_2 \geq 1.00000008 \tag{16}$$

$$\tag{17}$$

The primal simplex method concludes infeasibility during the presolve:

---

**Iteration Log #6**

```
CPLEX> primopt
Infeasibility row 'c1':  0 <= -1.92e-06.
Presolve time =    0.00 sec.
Presolve - Infeasible.
Solution time =    0.00 sec.
```

---

Turning presolve off causes the primal simplex method to arrive at a similar conclusion during the first iteration.

---

**Iteration Log #7**

```
Primal simplex - Infeasible:  Infeasibility =  1.9199999990e-06
Solution time =    0.00 sec.  Iterations = 0 (0)
CPLEX> display solution reduced -
Variable Name             Reduced Cost
x1                           -1.000000
x2                           24.000000
CPLEX> display solution slacks -
Constraint Name            Slack Value
slack c1                     -0.000002**
CPLEX> display solution basis variables -
There are no basic variables in the given range.
CPLEX> display solution basis slack -
Constraint 'c1' is basic.
```

---

The asterisks on the slack value for constraint $c1$ signify that the solution violates the slack's lower bound of 0.

These two runs both constitute correct outcomes. In Iteration Log #6, CPLEX's presolve uses the variable bounds and constraint coefficients to calculate that the minimum possible value for

the left hand side of constraint $c1$ is $-3 + 24 * 1.00000008 = 21 + 1.92 * 10^{-6}$. This means that the left hand side must exceed the right hand side, and by a value of more than that of CPLEX's default feasibility tolerance of $10^{-6}$. Iteration Log #7 shows that with presolve off, CPLEX begins the primal simplex method with the slack on constraint $c1$ in the basis, and the variables $x_1$ and $x_2$ at their respective bounds of 3 and 1.00000008. Given this basis, the reduced costs, i.e., the optimality criterion from Phase I, indicate that there is no way to remove the infeasibility, so the primal simplex method declares the model infeasible. Note that most optimizers treat variable bound constraints separately from general linear constraints, and that a negative reduced cost on a variable at its upper bound such as $x_1$ indicates that decreasing that variable from its upper bound cannot decrease the objective. Now, suppose we run the primal simplex method with a starting basis of $x_2$, the slack variable nonbasic at its lower bound, and $x_1$ nonbasic at its upper bound. The resulting basic solution of $x_1 = 3$, $x_2 = 1$, slack on $c1 = 0$ satisfies constraint c1 exactly. The variable $x_2$ does not satisfy its lower bound of 1.00000008 exactly, but the violation is less than many optimizers' default feasibility tolerance of $10^{-6}$. So, with this starting basis, an optimizer could declare the model feasible (and hence optimal, because the model has no objective function):

---

**Iteration Log #8**

```
Primal simplex - Optimal:  Objective = 0.0000000000e+00
Solution time = 0.00 sec.  Iterations = 0 (0)
```

---

In this example, were we to set the feasibility tolerance to $10^{-9}$, we would have obtained consistent results with respect to both bases because the data do not possess values smaller than the relevant algorithm tolerance. Although the value of .00000008 is input data, this small numerical value could have just as easily been created during the course of the execution of the algorithm. This example illustrates the importance of verifying that the optimizer tolerances properly distinguish legitimate values from those arising from round-off error. When a model is on the edge of feasibility, different bases may prove feasibility or infeasibility relative to the optimizer's tolerances. Rather than relying on the optimizer to make such important decisions, the practitioner should ensure that the optimizer's tolerances are suitably set to reflect the valid precision of the data values in the model. In the example we just examined, one should first determine whether the lower bound on $x_2$ is really 1.00000008, or if, in fact, the fractional part is round-off error in the data calculation and the correct lower bound is 1.0. If the former holds, the practitioner should set

the optimizer's feasibility and optimality tolerances to values smaller than .00000008. If the latter holds, the practitioner should change the lower bound to its correct value of 1.0 in the model. In this particular example, the practitioner may be inclined to deduce that the correct value for the lower bound on $x_2$ is 1.0, because all other data in the instance are integers. More generally, examination of the possible round-off error associated with the procedures used to calculate the input data may help to distinguish round-off error from meaningful values.

One particularly problematic source of round-off error in the data involves the conversion of single precision values to their double precision counterparts used by most optimizers. Precision for an IEEE single precision value is $6 * 10^{-8}$, which is almost as large as many of the important default optimizer tolerances. For example, CPLEX uses default feasibility and optimality tolerances of $10^{-6}$. So, simply *representing* a data value in single precision can introduce round-off error of at least $6 * 10^{-8}$, and additional single precision data calculations can increase the round-off error above the afore-mentioned optimizer tolerances. Hence, the optimizer may subsequently make decisions based on round-off error. Computing the data in double precision from the start will avoid this problem. If that is not possible, setting the optimizer tolerances to values that exceed the largest round-off error associated with the conversion from single to double precision provides an alternative.

All linear programming algorithms can suffer from numerical instability. In particular, the choice of primal or dual simplex algorithm does not affect the numerical stability of a problem instance because the LU factorizations are the same with either algorithm. However, the interior point algorithm is more susceptible to numerical stability problems because it tries to maintain an interior solution, yet as the algorithm nears convergence, it requires a solution on lower dimensional faces of the polyhedron, i.e., the boundary of the feasible region.

## 3.2 Degeneracy

Degeneracy in the simplex algorithm occurs when the value $\theta$ in the minimum ratio test in Step **5** of the simplex algorithm (see §2.1) is zero. This results in iterations in which the objective retains the same value, rather than improving. Highly degenerate LPs tend to be more difficult to solve using the simplex algorithm. Iteration Log #9 illustrates degeneracy: the nonoptimal objective does not change between iterations 5083 and 5968; therefore, the algorithm temporarily perturbs the right hand side or variable bounds to move away from the degenerate solution.

---

**Iteration Log #9**

```
Iter: 4751 Infeasibility = 8.000000
Iter: 4870 Infeasibility = 8.000000
Iter: 4976 Infeasibility = 6.999999
Iter: 5083 Infeasibility = 6.000000
Iter: 5191 Infeasibility = 6.000000
...
Iter: 5862 Infeasibility = 6.000000
Iter: 5968 Infeasibility = 6.000000
Perturbation started.
```

---

After the degeneracy has been mitigated, the algorithm removes the perturbation to restore the original problem instance. If the current solution is not feasible, the algorithm performs additional iterations to regain feasibility before continuing the optimization run. Although a pricing scheme such as Bland's rule can be used to mitigate cycling through bases under degeneracy, this rule holds more theoretical, than practical, importance and, as such, is rarely implemented in state-of-the-art optimizers. While such rules prevent cycles of degenerate pivots, they do not necessarily prevent long sequences of degenerate pivots that do not form a cycle, but do inhibit primal or dual simplex method performance.

When an iteration log indicates degeneracy, first consider trying all other LP algorithms. Degeneracy in the primal LP does not necessarily imply degeneracy in the dual LP. Therefore, the dual simplex algorithm might effectively solve a highly primal degenerate problem, and vice versa. Interior point algorithms are not prone to degeneracy because they do not pivot from one extreme point to the next. Interior point solutions are, by definition, nondegenerate. If alternate algorithms do not help performance (perhaps due to other problem characteristics that make them disadvantageous), a small, random perturbation of the problem data may help. Primal degenerate problems can benefit from perturbations of the right hand side values, while perturbations of the objective coefficients can help on dual degenerate problems. While such perturbations do not guarantee that the simplex algorithm does not cycle, they frequently yield improvements in practical performance. Some optimizers allow the practitioner to request perturbations by setting a parameter; otherwise, one can perturb the problem data explicitly.

### 3.3 Excessive Simplex Algorithm Iteration Counts

As described in the previous section, degeneracy can increase simplex algorithm iteration counts. However, the simplex algorithm may exhibit excessive iterations (typically, at least three times the number of constraints) for other reasons as well. For some models, the algorithm may make inferior choices when selecting the entering basic variable. In such cases, more computationally elaborate selection schemes than partial or full pricing that compute additional information can reduce the number of iterations enough to outweigh any associated increase in time per iteration. Today's state-of-the-art optimizers typically offer parameter settings that determine the entering variable selection to the practitioner, and selections other than the default can significantly improve performance.

Steepest edge and Devex pricing are the most popular of these more informative selection rules. Steepest edge pricing computes the L2 norm of each nonbasic matrix column relative to the current basis. Calculating this norm explicitly at each iteration by performing a forward solve, as in Step **4** of the primal simplex algorithm, would be prohibitively expensive (with the possible exception of when a large number of parallel threads is available). However, such computation is unnecessary, as all of the Steepest edge norms can be updated at each iteration with two additional backward solves, using the resulting vectors in inner products with the nonbasic columns in a manner analogous to full pricing (Goldfarb and Reid (1977) and Greenberg and Kalan (1975)). Devex pricing (Harris, 1975) estimates part of the Steepest edge update, removing one of the backward solves and one of the aforementioned inner products involving the nonbasic matrix columns. These methods can be implemented efficiently in both the primal and dual simplex algorithms.

The initial calculation of the exact Steepest edge norms can also be time consuming, involving a forward solve for each L2 norm or a backward solve for each constraint. Steepest edge variants try to reduce or remove this calculation by using estimates of the initial norms that are easier to compute. After computing these initial estimates, subsequent updates to the norms are done exactly. By contrast, Devex computes initial estimates to the norms, followed by estimates of the norm updates as well.

Since the additional computations for Steepest edge comprise almost as much work as a primal simplex algorithm iteration, this approach may need a reduction in the number of primal simplex iterations of almost 50 percent to be advantageous for the algorithm. However, the Steepest edge norm updates for the dual simplex algorithm involve less additional computational expense (Goldfarb and Forrest, 1992); in this case, a 20 percent reduction in the number of iterations may suffice to improve performance. Devex pricing can also be effective if it reduces iteration counts by 20 percent or more. State-of-the-art optimizers may already use some form of Steepest edge pricing

by default. If so, Steepest edge variants or Devex, both of which estimate initial norms rather than calculate them exactly, may yield similar iteration counts with less computation time per iteration.

Model characteristics such as constraint matrix density and scaling typically influence the trade-off between the additional computation time and the potential reduction in the number of iterations associated with Steepest edge, Devex or other entering variable selection schemes. The practitioner should keep this in mind when assessing the effectiveness of these schemes. For example, since these selection rules involve additional backward solves and pricing operations, their efficacy depends on the density of the constraint matrix $A$. Denser columns in $A$ increase the additional computation time per iteration. Some optimizers have default internal logic to perform such assessments automatically and to use the selection scheme deemed most promising. Nonetheless, for LPs with excessive iteration counts, trying these alternate variable selection rules can improve performance relative to the optimizer's default settings.

Iteration Logs #6, #7 and #8 illustrate the importance of selecting optimizer tolerances to properly distinguish legitimate values from those arising from round-off error. In those iteration logs, this distinction was essential to determine if a model was infeasible or feasible. This distinction can also influence the number of simplex algorithm iterations, and proper tolerance settings can improve performance. In particular, many implementations of the simplex method use the Harris ratio test (Harris, 1975) or some variant thereof. Harris' method allows more flexibility in the selection of the outgoing variable in the ratio test for the primal or dual simplex method, but it does so by allowing the entering variable to force the outgoing variable to a value slightly below 0. The optimizer then shifts the variable lower bound of 0 to this new value to preserve feasibility. These bound shifts are typically limited to a tolerance value no larger than the optimizer's feasibility tolerance. While this offers advantages regarding more numerically stable pivots and less degeneracy, such violations must eventually be addressed since they can potentially create small infeasibilities. In some cases, performance can be improved by reducing the maximum allowable violation in the Harris ratio test.

Iteration Logs #10 and #11 illustrate how reducing the maximum allowable violation in the Harris ratio test can improve performance. The model, pilot87, is publicly available from the NETLIB set of linear programs at `http://www.netlib.org/lp/`. Iteration Log #10 illustrates a run with default feasibility and optimality tolerances of $10^{-6}$. However, because pilot87 contains matrix coefficients as small as $10^{-6}$, the resulting bound violations and shifts allowed in the Harris ratio test can create meaningful infeasibilities. Hence, when CPLEX removes the shifted bounds starting at iteration $9,161$, it must repair some modest dual infeasibilities with additional dual simplex iterations. Subsequent iterations are performed with a reduced limit on the bound shift,

but the removal of the additional bound shifts at iteration 10,008 results in some slight dual infeasibilities requiring additional iterations. Overall, CPLEX spends 993 additional dual simplex iterations after the initial removal of bound shifts at iteration 9,161. Since this model has (apparently legitimate) matrix coefficients of $10^{-6}$, the default feasibility and optimality tolerances are too large to enable the optimizer to properly distinguish legitimate values from round-off error. The bound shifts thus are large enough to create dual infeasibilities that require additional dual simplex iterations to repair.

**Iteration Log #10**

```
Problem '/ilog/models/lp/all/pilot87.sav.gz' read.

...

Iteration log . . .
Iteration:     1   Scaled dual infeas =             0.676305
Iteration:   108   Scaled dual infeas =             0.189480
Iteration:   236   Scaled dual infeas =             0.170966
...
Iteration:  8958   Dual objective     =           302.913722
Iteration:  9056   Dual objective     =           303.021157
Iteration:  9137   Dual objective     =           303.073444
Removing shift (4150).
Iteration:  9161   Scaled dual infeas = 0.152475
Iteration:  9350   Scaled dual infeas =             0.001941
Iteration:  9446   Scaled dual infeas =             0.000480
Iteration:  9537   Dual objective     =           299.891447
Iteration:  9630   Dual objective     =           301.051704
Iteration:  9721   Dual objective     =           301.277884
Iteration:  9818   Dual objective     =           301.658507
Iteration:  9916   Dual objective     =           301.702665
Removing shift (41).
Iteration: 10008   Scaled dual infeas = 0.000136
Iteration: 10039   Dual objective     =           301.678880
Iteration: 10140   Dual objective     =           301.710360
```

```
Removing shift (8).
Iteration: 10151    Objective     =           301.710354

Dual simplex - Optimal:  Objective =  3.0171035068e+02
Solution time =    6.38 sec.  Iterations = 10154 (1658)
```

By contrast, Iteration Log #11 illustrates the corresponding run with feasibility and optimality tolerances reduced to $10^{-9}$. This enables CPLEX to distinguish the matrix coefficient of $10^{-6}$ as legitimate in the model. Thus, it uses smaller bound violations and shifts in the Harris ratio test. Therefore, it needs to remove the bound shifts once, and only requires 32 additional dual simplex iterations to prove optimality. While the overall reduction in run time with this change is a modest four percent, larger improvements are possible on larger or more numerically challenging models.

**Iteration Log #11**

```
New value for feasibility tolerance: 1e-09
New value for reduced cost optimality tolerance: 1e-09

...

Iteration log . . .
Iteration:     1   Scaled dual infeas =             0.676355
Iteration:   123   Scaled dual infeas =             0.098169
...
Removing shift (190).
Iteration:  9332   Scaled dual infeas = 0.000004
Iteration:  9338   Dual objective     =           301.710248

Dual simplex - Optimal:  Objective =  3.0171034733e+02
Solution time =    6.15 sec.  Iterations = 9364 (1353)
```

## 3.4 Excessive Barrier Algorithm Iteration Counts

Section 2.3 included a discussion of how the non-zero structure of the constraint matrix influences barrier and other interior point algorithms' time per iteration. On most LPs, the weakly polynomial complexity of the barrier algorithm results in a very modest number of iterations, even

as the size of the problem increases. Models with millions of constraints and variables frequently solve in fewer than 100 iterations. However, because the barrier algorithm relies on convergence criteria, the algorithm may struggle to converge, performing numerous iterations with little or no improvement in the objective. Most barrier algorithm implementations include an adjustable convergence tolerance that can be used to determine when the algorithm should stop, and proceed to the crossover procedure to find a basic solution. For some models, increasing the barrier convergence tolerance avoids barrier iterations of little, if any, benefit to the crossover procedure. In such cases, a larger barrier convergence tolerance may save significant time when the barrier method run time dominates the crossover run time. By contrast, if the crossover time with default settings comprises a significant part of the optimization time and the barrier iteration count is modest, reducing the barrier convergence tolerance may provide the crossover procedure a better interior point with which to start, thus improving performance.

Iteration logs #12 and #13 provide an example in which increasing the barrier convergence tolerance improves performance. The model solved in the logs is Linf_520c.mps, publicly available at Hans Mittelmann's Benchmarks for Optimization Software website (`http://plato.asu.edu/ftp/lptestset/`). Because this model has some small coefficients on the order of $10^{-6}$, the runs in these logs follow the recommendations in Section 3.1 and use feasibility and optimality tolerances of $10^{-8}$ to distinguish them from any meaningful values in the model.

Columns 2 and 3 of Iteration Log #12 provide the information needed to assess the relative duality gap $\frac{(c^T x - y^T b)}{c^T x}$ typically compared with the barrier convergence tolerance. Not surprisingly, the gap is quite large initially. However, the gap is much smaller by iteration 20. Modest additional progress occurs by iteration 85, but little progress occurs after that, as can be seen at iterations 125 and 175. In fact, at iteration 175, CPLEX's barrier algorithm has stopped improving relative to the convergence criteria, as can be seen by the slight increases in relative duality gap, primal bound infeasibility and dual infeasibility (in the second, third, fifth and sixth columns of the iteration log). CPLEX therefore initiates crossover beginning at iteration 19. Thus, iterations 20 through 175 are essentially wasted. The solution values at iteration 19 still have a significant relative duality gap, so the crossover procedure finds a basis that requires additional simplex method iterations. Iteration Log #13 illustrates how, by increasing the barrier convergence tolerance from the CPLEX default of $10^{-8}$ to $10^{-4}$, much of the long tail of essentially wasted iterations is removed. In this case, the optimizer does not need to restore solution values from an earlier iteration due to increases in the relative duality gap. Hence, the optimizer initiates crossover at a solution closer to optimality. The basis determined by crossover is much closer to optimal than the one in Iteration Log #12, resulting in very few additional simplex iterations to find an optimal basis. This reduction in

simplex iterations, in addition to the reduced number of barrier iterations, improves the overall run time from $1,063$ seconds to 452 seconds.

By contrast, if the barrier iteration counts are small (e.g., 50 or fewer), show no long tail of iterations with little progress (as in Log #12), yet exhibit significant crossover and simplex iterations, decreasing the barrier convergence tolerance, rather than increasing it, often improves performance.

**Iteration Log #12**

| Itn | Primal Obj | Dual Obj | Prim Inf | Upper Inf | Dual Inf |
|---|---|---|---|---|---|
| 0 | 5.9535299e+02 | -2.7214222e+13 | 4.05e+11 | 1.03e+12 | 3.22e+07 |
| 1 | 1.8327921e+07 | -2.4959566e+13 | 2.07e+10 | 5.30e+10 | 6.74e+04 |
| 2 | 7.5190135e+07 | -5.6129961e+12 | 4.68e+09 | 1.20e+10 | 1.09e+03 |
| 3 | 2.8110662e+08 | -2.2938869e+12 | 5.95e+07 | 1.52e+08 | 4.46e+02 |
| 4 | 2.8257248e+08 | -4.7950310e+10 | 1.28e+06 | 3.26e+06 | 3.53e+00 |
| ... | | | | | |
| 19 | 2.0017972e-01 | 1.9840302e-01 | 1.37e-05 | 1.14e-06 | 1.51e-06 |
| 20 | 2.0003146e-01 | 1.9848712e-01 | 1.73e-05 | 1.30e-06 | 1.65e-06 |
| 21 | 1.9987509e-01 | 1.9854536e-01 | 1.62e-05 | 1.37e-06 | 1.71e-06 |
| ... | | | | | |
| 80 | 1.9896330e-01 | 1.9865235e-01 | 1.35e-04 | 2.87e-06 | 2.33e-06 |
| 81 | 1.9895653e-01 | 1.9865287e-01 | 1.50e-04 | 2.79e-06 | 2.32e-06 |
| 82 | 1.9895461e-01 | 1.9865256e-01 | 1.52e-04 | 2.87e-06 | 2.44e-06 |
| 83 | 1.9895075e-01 | 1.9865309e-01 | 1.69e-04 | 2.86e-06 | 2.38e-06 |
| 84 | 1.9894710e-01 | 1.9865303e-01 | 1.70e-04 | 2.85e-06 | 2.39e-06 |
| 85 | 1.9894158e-01 | 1.9865347e-01 | 1.68e-04 | 2.77e-06 | 2.40e-06 |
| ... | | | | | |
| 123 | 1.9888124e-01 | 1.9865508e-01 | 4.81e-04 | 3.01e-06 | 2.31e-06 |
| 124 | 1.9888029e-01 | 1.9865515e-01 | 4.46e-04 | 2.89e-06 | 2.37e-06 |
| 125 | 1.9887997e-01 | 1.9865506e-01 | 4.35e-04 | 2.93e-06 | 2.43e-06 |

```
...

 170   1.9886958e-01   1.9865768e-01  9.10e-04  2.63e-06  2.37e-06
 171   1.9886949e-01   1.9865771e-01  8.96e-04  2.56e-06  2.33e-06
 172   1.9886939e-01   1.9865739e-01  8.15e-04  2.45e-06  2.28e-06
 173   1.9886934e-01   1.9865738e-01  8.19e-04  2.47e-06  2.26e-06
 174   1.9886931e-01   1.9865728e-01  7.94e-04  2.52e-06  2.29e-06
 175   1.9886927e-01   1.9865716e-01  7.57e-04  2.55e-06  2.31e-06
   *   2.0017972e-01   1.9840302e-01  1.37e-05  1.14e-06  1.51e-06
Barrier time =  869.96 sec.

Primal crossover.
  Primal:  Fixing 34010 variables.
    34009 PMoves:  Infeasibility  3.11238869e-08  Objective  2.00179717e-01
    32528 PMoves:  Infeasibility  4.93933590e-08  Objective  2.00033745e-01
    31357 PMoves:  Infeasibility  6.39691687e-08  Objective  2.00033745e-01

...

  Elapsed crossover time =   20.14 sec. (3600 PMoves)
     3021 PMoves:  Infeasibility  1.14510022e-07  Objective  2.00033487e-01
     2508 PMoves:  Infeasibility  9.65797950e-08  Objective  2.00033487e-01

 ...

  Primal:  Pushed 15282, exchanged 18728.
  Dual:  Fixing 15166 variables.
    15165 DMoves:  Infeasibility  1.03339605e+00  Objective  1.98870673e-01
   Elapsed crossover time =   27.38 sec. (14800 DMoves)

...

   Elapsed crossover time =   68.41 sec. (1400 DMoves)
```

```
        0 DMoves:  Infeasibility  1.00261077e+00  Objective  1.98865022e-01
  Dual:  Pushed 10924, exchanged 4.

...

Iteration log . . .
Iteration:     1    Scaled infeas =         28211.724670
Iteration:    31    Scaled infeas =         23869.881089
Iteration:   312    Scaled infeas =          4410.384413

...

Iteration:  6670    Objective     =             0.202403
Iteration:  6931    Objective     =             0.199894
Elapsed time = 1061.69 sec. (7000 iterations).
Removing shift (5).
Iteration:  7072    Scaled infeas =             0.000000
Total crossover time =  192.70 sec.

Total time on 4 threads = 1063.55 sec.

Primal simplex - Optimal:  Objective =  1.9886847000e-01
Solution time = 1063.55 sec.  Iterations = 7072 (5334)
```

**Iteration Log #13**

| Itn | Primal Obj | Dual Obj | Prim Inf | Upper Inf | Dual Inf |
|---|---|---|---|---|---|
| 0 | 5.9535299e+02 | -2.7214222e+13 | 4.05e+11 | 1.03e+12 | 3.22e+07 |
| 1 | 1.8327921e+07 | -2.4959566e+13 | 2.07e+10 | 5.30e+10 | 6.74e+04 |
| 2 | 7.5190135e+07 | -5.6129961e+12 | 4.68e+09 | 1.20e+10 | 1.09e+03 |
| 3 | 2.8110662e+08 | -2.2938869e+12 | 5.95e+07 | 1.52e+08 | 4.46e+02 |
| 4 | 2.8257248e+08 | -4.7950310e+10 | 1.28e+06 | 3.26e+06 | 3.53e+00 |
| 5 | 1.3900254e+08 | -6.1411363e+07 | 1.21e-03 | 9.44e-07 | 6.00e-02 |

...

```
  82   1.9895461e-01   1.9865256e-01  1.52e-04  2.87e-06  2.44e-06
  83   1.9895075e-01   1.9865309e-01  1.69e-04  2.86e-06  2.38e-06
  84   1.9894710e-01   1.9865303e-01  1.70e-04  2.85e-06  2.39e-06
Barrier time =  414.31 sec.

Primal crossover.
  Primal:  Fixing 33950 variables.
    33949 PMoves:  Infeasibility  4.36828532e-07  Objective  1.98868436e-01
    32825 PMoves:  Infeasibility  4.32016730e-07  Objective  1.98868436e-01

...

      128 PMoves:  Infeasibility  5.49582923e-07  Objective  1.98868436e-01
        0 PMoves:  Infeasibility  5.49534369e-07  Objective  1.98868436e-01
  Primal:  Pushed 13212, exchanged 20737.
  Dual:  Fixing 71 variables.
       70 DMoves:  Infeasibility  2.72300071e-03  Objective  1.98842990e-01
        0 DMoves:  Infeasibility  2.72073853e-03  Objective  1.98842990e-01
  Dual:  Pushed 63, exchanged 0.
Using devex.

Iteration log . . .
Iteration:     1    Objective     =             0.198868
Removing shift (45).
Iteration:     2    Scaled infeas =             0.000000
Iteration:     6    Objective     =             0.198868
Total crossover time =   38.39 sec.

Total time on 4 threads =  452.71 sec.

Primal simplex - Optimal:  Objective =  1.9886847000e-01
Solution time =  452.71 sec.  Iterations = 6 (4)
```

## 3.5 Excessive Time per Iteration

Algorithms may require an unreasonable amount of time per iteration. For example, consider Iteration Log #14 in which 37,000 iterations are executed within the first 139 seconds of the run; at the bottom of the log, 1000 (140,000 - 139,000) iterations require about 408 (25,145.43 - 24,736.98) seconds of computation time.

---

**Iteration Log #14**

```
Elapsed time = 138.23 sec. (37000 iterations)
Iter: 37969 Infeasibility = 387849.999786
Iter: 39121 Infeasibility = 379979.999768
Iter: 40295 Infeasibility = 375639.999998
Elapsed time = 150.41 sec. (41000 iterations)
...
Elapsed time = 24318.58 sec. (138000 iterations)
Iter: 138958 Infeasibility = 23.754244
Elapsed time = 24736.98 sec. (139000 iterations)
Elapsed time = 25145.43 sec. (140000 iterations)
```

---

This slowdown in iteration execution can be due to denser bases and the associated denser factorization and solve times in the simplex algorithm, specifically in Steps **1**, **4** and **7**. In this case, the practitioner should try each LP algorithm and consider an alternate pricing scheme if using the simplex algorithm. The more computationally expensive gradient pricing schemes available in today's state-of-the-art optimizers calculate (exactly or approximately) the norm of each nonbasic matrix column relative to the current basis. The most expensive calculations involve computing these norms exactly, while less expensive calculations involve progressively cheaper (but progressively less accurate) ways of estimating these exact norms (Nazareth, 1987). All of these norm calculations involve extra memory, extra computation, and extra pricing operations.

Another reason for excessive time per iteration is the computer's use of virtual memory. A general rule requires one gigabyte of memory per million constraints of a linear program and even more if the $A$ matrix is very dense or if there is a significantly larger number of columns (variables) than rows (constraints). Short of purchasing more memory, some optimizers support non-default settings that compress data and/or store data efficiently to disk. For example, CPLEX has a

memory emphasis parameter whose overhead is mitigated by removing the optimizer's reliance on the operating system's virtual memory manager. This can help preserve memory and ultimately lead to finding a good solution reasonably quickly.

While the change in density of the basis matrices used by the simplex method can alter the time per iteration, each iteration of the barrier algorithm and related interior point methods solves a linear system involving $AA^T$ with constant non-zero structure. Hence, barrier algorithm memory usage and time per iteration typically exhibit little change throughout the run. Section 2.3 describes how the density of $AA^T$ and the associated Cholesky factorization influence the time per barrier algorithm iteration. However, that section assumed that the density of the Cholesky factor was given, and identified guidelines for assessing when the density was favorable for the barrier algorithm. While state-of-the-art optimizers do offer some parameter settings that can potentially reduce the density of the Cholesky factor, the default settings are very effective regarding the sparsity of the Cholesky factor for a given non-zero structure of $AA^T$. Instead of trying to find a sparser Cholesky factor for a given $AA^T$, the practitioner can adjust the model formulation so that $AA^T$ is sparser, resulting in a sparser Cholesky factor and faster barrier algorithm iterations. One example of reformulation involves splitting dense columns into sparser ones (Lustig et al. (1991) and Vanderbei (1991)). While this increases the number of constraints and variables in the model, those dimensions are not the source of the performance bottleneck. A larger model with a Cholesky factor that has more constraints but fewer non-zeros can result in faster performance. For example, if $A_j$ is a dense matrix column associated with variable $x_j$, it can be split into two (or more) sparser columns $A_j^1$ and $A_j^2$ that, when combined, intersect the same rows as $A_j$. By defining $A_j^1$ and $A_j^2$ so that $A_j = A_j^1 + A_j^2$, and their non-zero indices do not intersect, a single dense column can be replaced by two sparser ones. In the model formulation, $A_j x_j$ is replaced by $A_j^1 x_j^1 + A_j^2 x_j^2$. The model also requires an additional constraint $x_j^1 - x_j^2 = 0$. The value of $x_j$ in the final solution is $x_j^1$ (or $x_j^2$).

Column splitting increases the problem size to improve performance. However, this somewhat counterintuitive approach can be effective because the reduction in run time associated with the dense columns exceeds the increase in run time associated with additional variables and constraints. More generally, if an increase in problem size removes a performance problem in an algorithm without creating a new bottleneck, it may improve overall algorithm performance.

Table 3 summarizes problems and suggestions for resolving them as addressed in Section 3. These suggestions assume that the practitioner has formulated the model in question with the correct fidelity, i.e., that the model cannot be reduced without compromising the value of its solution. When the model size becomes potentially problematic, as illustrated in Iteration Logs #2

and #14, the practitioner should also consider whether solving a less refined, smaller model would answer his needs.

| LP performance issue | Suggested resolution |
|---|---|
| Numerical instability | Calculate and input model data in double precision<br>Eliminate nearly-redundant rows and/or columns of $A$ *a priori*<br>Avoid mixtures of large and small numbers:<br>($i$) Be suspicious of $\kappa$ between $10^{10}$ and $10^{14}$;<br>($ii$) Avoid data leading to $\kappa$ greater than $10^{14}$<br>Use alternate scaling (in the model formulation or optimizer settings)<br>Increase the Markowitz threshold<br>Employ the numerical emphasis parameter (if available) |
| Lack of objective function improvement under degeneracy | Try all other algorithms (and variants)<br>Perturb data either a priori or using algorithmic settings |
| Primal degeneracy | Use either dual simplex or interior point on primal problem |
| Dual degeneracy | Employ either primal simplex or interior point on primal problem |
| Both primal and dual degeneracy | Execute interior point on primal or dual problem |
| Excessive time per iteration | Try all other algorithms (and variants)<br>Use algorithmic settings to conserve memory or purchase more externally<br>Try less expensive pricing settings if using simplex algorithms |
| Excessive simplex algorithm iterations | Try Steepest edge or Devex variable selection |
| Multiple bound shift removals or significant infeasibilities after removing shifts | Reduce feasibility and optimality tolerances |
| Barrier algorithm iterations with little or no progress | Increase barrier convergence tolerance in order to initiate crossover earlier |
| Too much time in crossover | Reduce barrier convergence tolerance in order to provide better starting point for crossover |

Table 3: LP Performance issues and their suggested resolution

When neither the tactics in Table 3 nor any other tuning of the simplex method or barrier algorithms yields satisfactory performance, the practitioner may also consider more advanced variants of the simplex method. Such variants frequently solve a sequence of easier, more tractable LPs, resulting in an optimal solution to the more difficult LP of interest. Some variants, such as sifting (also called the SPRINT technique by its originator, John Forrest (Forrest, 1989)), work on parts of the LP of interest, solving a sequence of LPs in which the solution of the previous one defines problem modifications for the next one. In particular, sifting works well on LPs in which the number of variables dramatically exceeds the number of constraints. It starts with an LP consisting of all constraints but a manageable subset of the variables. After solving this LP subproblem, it uses the optimal dual variables to compute reduced costs that identify potential

variables to add. The basis from the previous LP is typically primal feasible, speeding up the next optimization. Through careful management of added and removed variables from the sequence of LPs it solves, sifting can frequently solve the LP of interest to optimality without ever having to represent it in memory in its entirety. Sifting can also work well on models for which the number of constraints dramatically exceeds the number of variables by applying sifting to the dual LP. Other variants are more complex, maintaining two or more distinct subproblems, and using the solution of one to modify the other. The most frequently used methods are Dantzig-Wolfe Decomposition and Benders' Decomposition. These methods both maintain a master problem and solve a separate subproblem to generate modifications to the master problem. Both the master and subproblem are typically much easier to solve than the original LP of interest. Repeatedly optimizing the master and subproblem ultimately yields an optimal solution to the original problem. Dantzig-Wolfe Decomposition is column-based, using the subproblem to generate additional columns for the master problem, while Benders' Decomposition is row-based, solving the subproblem to generate additional rows for the master problem. It can be shown that Benders' Decomposition applied to the primal representation of an LP is equivalent to applying Dantzig-Wolfe Decomposition to the dual LP. The details and application of these and other decomposition methods are beyond the scope of this paper. Dantzig and Thapa (2003) provide more information about these approaches.

# 4 Conclusions

We summarize in this paper commonly employed linear programming algorithms and use this summary as a basis from which we present likely algorithmic trouble and associated avenues for their resolution. While optimizers and hardware will continue to advance in their capabilities of handling hard linear programs, practitioners will take advantage of corresponding improved performance to further refine their models. The guidelines we present are useful, regardless of the anticipated advances in hardware and software. Practitioners can implement many of these guidelines without expert knowledge of the underlying theory of linear programming, thereby enabling them to solve larger and more detailed models with existing technology.

# Acknowledgements

Dr. Klotz wishes to acknowledge all of the CPLEX practitioners over the years, many of whom have provided the wide variety of models that revealed the guidelines described in this paper. He also wishes to thank the past and present CPLEX development, support, and sales and marketing teams who have contributed to the evolution of the product. Professor Newman wishes to thank

the students in her first advanced linear programming class at the Colorado School of Mines for their helpful comments; she also wishes to thank her colleagues Professor Josef Kallrath (BASF-AG, Ludwigshafen, Germany) and Jennifer Rausch (Jeppeson, Englewood, Colorado) for helpful comments on an earlier draft. Both authors thank an anonymous referee for his helpful comments that lead to the improvement of the paper.

Both authors also wish to remember Lloyd Clarke (February 14, 1964-September 20, 2007). His departure from the CPLEX team had consequences that extended beyond the loss of an important employee and colleague.

# References


Applegate, D., Cook, W. and Dash, S., 2005. "QSopt." `http://www.isye.gatech.edu/~wcook/qsopt/`.

Bazaraa, M., Jarvis, J. and Sherali, H., 2005. *Linear Programming and Network Flows*, John Wiley & Sons, Inc.

Bertsimas, D. and Tsitsiklis, J., 1997. *Introduction to Linear Optimization*, Prentice Hall, chap. 4.

Brown, G. and Rosenthal, R., 2008. "Optimization tradecraft: Hard-won insights from real-world decision support." *Interfaces*, **38**(5): 356–366.

Chvátal, V., 1983. *Linear Programming*, W. H. Freeman.

Cybernet Systems Co., 2012. "Maple." http://www.maplesoft.com/.

Dantzig, G., 1963. *Linear Programming and Extensions*, Princeton University Press.

Dantzig, G. and Thapa, M., 1997. *Linear Programming 1: Introduction*, Springer.

Dantzig, G. and Thapa, M., 2003. *Linear Programming 2: Theory and Extensions*, Springer.

Dikin, I., 1967. "Iterative solution of problems of linear and quadratic programming." *Soviet Mathematics Doklady*, **8**: 674–675.

Duff, I., Erisman, A. and Reid, J., 1986. *Direct Methods for Sparse Matrices*, Clarendon Press Oxford.

Fiacco, A. and McCormick, G., 1968. *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*, Wiley.

FICO, 2012. *Xpress-MP Optimization Suite*, Minneapolis, MN.

Forrest, J., 1989. "Mathematical programming with a library of optimization routines." Presentation at 1989 ORSA/TIMS Joint National Meeting.

Fourer, B., 1994. "Notes on the dual simplex method." `http://users.iems.northwestern.edu/~4er/WRITINGS/dual.pdf`.

Gill, P., Murray, W., Saunders, M., Tomlin, J. and Wright, M., 1986. "On projected Newton barrier methods for linear programming and an equivalence to Karmarkar's projective method." *Mathematical Programming*, **36**(2): 183–209.

Goldfarb, D. and Forrest, J., 1992. "Steepest-edge simplex algorithms for linear programming." *Mathematical Programming*, **57**(1): 341–374.

Goldfarb, D. and Reid, J., 1977. "A practical steepest-edge simplex algorithm." *Mathematical Programming*, **12**(1): 361–371.

Greenberg, H. and Kalan, J., 1975. "An exact update for Harris' TREAD." *Mathematical Programming Study*, **4**(1): 26–29.

Gurobi, 2012. *Gurobi Optimizer*, Houston, TX.

997 Harris, P., 1975. "Pivot selection methods in the Devex LP code." *Mathematical Programming*
998 *Study*, **4**(1): 30–57.

999 Higham, N., 1996. *Accuracy and Stability of Numerical Algorithms*, SIAM.

1000 IBM, 2012. *ILOG CPLEX*, Incline Village, NV.

1001 Karmarkar, N., 1984. "A new polynomial-time algorithm for linear programming." *Combinatorica*,
1002 **4**(4): 373–395.

1003 Khachian, L., 1979. "A polynomial algorithm for linear programming." *Soviet Mathematics Dok-*
1004 *lady*, **20**: 191–194.

1005 Lemke, C., 1954. "The dual method of solving the linear programming problem." *Naval Research*
1006 *Logistics Quarterly*, **1**(1): 36–47.

1007 Lustig, I., Marsten, R., Saltzman, M., Subramanian, R. and Shanno, D., 1990. "Interior-point
1008 methods for linear programming: Just call Newton, Lagrange, and Fiacco and McCormick!"
1009 *Interfaces*, **20**(4): 105–116.

1010 Lustig, I., Marsten, R. and Shanno, D., 1994. "Interior-point methods for linear programming:
1011 Computational state of the art." *ORSA Journal on Computing*, **6**(1): 1–14.

1012 Lustig, I., Mulvey, J. and Carpenter, T., 1991. "Formulating two-stage stochastic programs for
1013 interior point methods." *Operations Research*, **39**(5): 757–770.

1014 Megiddo, N., 1991. "On finding primal- and dual- optimal bases." *ORSA Journal on Computing*,
1015 **3**(1): 63–65.

1016 Mehrotra, S., 1992. "On the implementation of a primal-dual interior point method." *SIAM Journal*
1017 *on Optimization*, **2**(4): 575–601.

1018 MOPS, 2012. *MOPS*, Paderborn, Germany.

1019 MOSEK, 2012. *MOSEK Optimization Software*, Copenhagen, Denmark.

1020 Nazareth, J., 1987. *Computer Solution of Linear Programs*, Oxford University Press.

1021 Rardin, R., 1998. *Optimization in Operations Research*, Prentice Hall, chap. 6.

1022 Vanderbei, R., 1991. "Splitting dense columns in sparse linear systems." *Linear Algebra and its*
1023 *Applications*, **152**(1): 107–117.

1024 Winston, W., 2004. *Operations Research: Applications and Algorithms*, Brooks/Cole, Thompson
1025 Learning.

1026 Wolfram, 2012. "Mathematica." http://www.wolfram.com/mathematica/.

1027 Wright, S., 1997. *Primal-Dual Interior-Point Methods*, SIAM.