Annals of Management Science
Vol. 3, No. 1, May 2014, 43-64

Computing a hybrid preconditioner approach to solve the linear systems arising from interior point methods for linear programming using the conjugate gradient method

Carla T. L. S. Ghidini[a], A. R. L. Oliveira[a*] and D. C. Sorensen[b]
[a]University of Campinas (UNICAMP), 13083-859 Campinas - SP, Brazil
[b]Department of Computational and Applied Mathematics,Rice University Houston Texas, U.S.A.

Abstract

In this work, iterative methods are used to solve the linear systems of equations arising from interior point methods. Since these systems of equations are very ill-conditioned near a solution, the design of specially tailored preconditioners is an important implementation issue. On the other hand, the early linear systems of equations do not present the same features and it is advisable to adopt hybrid preconditioners that begin as a generic preconditioner and adapt during the course of the iteration, becoming ever more specialized as convergence takes place. During the initial iterations, a controlled Cholesky factorization is used. As convergence takes place, a splitting, the splitting preconditioner is adopted. Its major advantage is its excellent behavior near a solution of the linear program. This desirable feature has a price. The preconditioner could be very expensive to compute. A careful implementation must be performed in order to achieve competitive results regarding both: speed and robustness. An effective implementation of the splitting preconditioner relies upon finding a suitable set of linearly independent columns to form a nonsingular matrix from the constraint matrix. Several strategies to help finding such set of columns are presented. Numerical experiments are carried out in order to illustrate the performance of the given strategies.

Keywords: *linear programming, interior point methods, preconditioning.*

1. Introduction

The development of sophisticated software to solve linear optimization problems by interior point methods has started since the early works on this subject. There are three main research lines aimed at improving the efficiency of such methods for solving large-scale problems: reduction of the total number of iterations, techniques to obtain a fast iteration and specific methods for particular classes of problems.

[*] Corresponding author. Email: aurelio@ime.unicamp.br

This work addresses the second one. Iterative methods are used to solve the linear systems of equations which are the most expensive step at each iteration of interior point methods. Since such systems are very ill-conditioned near a solution, the design of specially tailored preconditioners is an important implementation issue. On the other hand, since the early linear systems do not present the same features, it is advisable to adopt hybrid preconditioners that begin as a generic preconditioner and adapt during the course of the iteration, becoming ever more specialized as convergence takes place (Bocanegra et al., 2007).

During the initial iterations a controlled Cholesky factorization is adopted (Campos & Birkett, 1998). Its major advantage is the control parameter that allows the preconditioner to vary all way from a diagonal preconditioner to the full Cholesky factorization, if desired. At the onset of convergence, a splitting preconditioner is used (Oliveira & Sorensen, 2005). Its major advantage is its excellent behavior near a solution of the linear program. However, this desirable feature has a price: the preconditioner could be very expensive to compute. A careful implementation must be performed in order to achieve competitive numerical results regarding both: speed and robustness. An effective implementation of the splitting preconditioner depends crucially upon finding a suitable set of linearly independent columns to form a nonsingular matrix, to be factored, from the constraint matrix.

There are several techniques for finding such a set of columns such as the delayed update form for the LU factorization, the symbolic dependent columns, the symbolic independent columns, the combination of symbolic dependent and independent columns and strongly connected components. Some are well known and already applied in other contexts (Coleman & Pothen, 1987; Duff & Reid, 1986; El-Bakry et al., 1994). Others were developed to compute the splitting preconditioner (Oliveira, 1997; Oliveira & Sorensen, 2005). Among the techniques used is the study of the nonzero structure of the constraint matrix to speed up the numerical factorization, such as using key columns, symbolically dependent and independent columns, finding strongly connected components (Oliveira & Sorensen, 2005). Other implementation issues, include ways for changing preconditioners, are also discussed in (Ghidini et al., 2012; Velazco et al., 2010).

The choice of the controlled factorization is justified due to the possibility of computing an inexpensive preconditioner in the initial interior point iterations and, as the linear systems become more ill conditioned, the controlled preconditioner can be improved with just the change of a parameter value. Numerical experiments illustrating the effectiveness of such strategies in order to solve large scale linear programming problems are presented in Bocanegra et al. (2007) and Velazco et al. (2010).

This work is organized as follows: Section 2 presents the predictor-corrector interior point method, defines its search directions and explains how to solve the resulting linear systems of equations. The controlled Cholesky factorization and splitting preconditioners are discussed in this section. Sections 3 and 4 study several techniques in order to achieve an efficient implementation of the splitting preconditioner. In Section 5 the numerical experiments are shown and discussed. Conclusions follow in Section 6.

2. Primal-Dual interior point methods

Consider the linear programming problem in the standard form:

$$\begin{array}{ll} \text{Min} & c^t x \\ \text{s.t.} & Ax = b\,, \\ & x \geq 0, \end{array} \tag{2.1}$$

where A is a full row rank $m \times n$ matrix and c, b and x are column vectors of appropriate dimension.

Associated with problem (2.1) is the dual linear programming problem

$$\begin{array}{ll} \text{Max} & b^t y \\ \text{s.t.} & A^t y + z = c, \\ & z \geq 0, \end{array} \tag{2.2}$$

where y is an m-vector of free variables and z is the n-vector of dual slack variables. The duality gap is defined as $c^t x$ - $b^t y$. It reduces to $x^t z$ for feasible points.

Since Karmarkar (1984) presented the first polynomial time interior point method for linear programming, many methods have appeared. One of the best among them is the predictor-corrector method (Mehrotra, 1992; Momoh et al., 1999). In the predictor-corrector approach, the search directions are obtained by solving two linear systems of equations by applying Newton's methods to the KKT conditions. First we compute the *affine directions*:

$$\begin{bmatrix} 0 & I & A^t \\ Z & X & 0 \\ A & 0 & 0 \end{bmatrix} . \begin{bmatrix} \Delta\tilde{x} \\ \Delta\tilde{z} \\ \Delta\tilde{y} \end{bmatrix} = \begin{bmatrix} r_d \\ r_a \\ r_p \end{bmatrix}, \tag{2.3}$$

where X = diag(x), Z = diag(z) and the residuals primal, dual and complementarity are given by: $r_p = b - Ax$, $r_d = c - A^t y - z$ and $r_a = -XZe$ and e is the vector of all ones. Then, the search directions (Δx, Δy, Δz) are computed solving (2.3) with r_a replaced by

$$r_c = \mu e - X\,Z\,e - \Delta\tilde{X}\Delta\tilde{Z}e\,,$$

where μ is the centering parameter.

Multiple corrections could be computed in order to improve the predictor corrector (Gondzio, 1996). Each additional direction is obtained by solving one linear system of equations with the matrix given above.

2.1. Computing the Search Directions

The computational cost at each iteration is dominated by the solution of linear systems such as (2.3). Since the systems share the same matrix, we will restrict the discussion to one linear system.

By eliminating the variables Δz the system reduces to:

$$\begin{bmatrix} -D & A^t \\ A & 0 \end{bmatrix} \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} r_d - X^{-1} r_a \\ r_p \end{bmatrix}, \tag{2.4}$$

where $D = X^{-1}Z$ is an $n \times n$ diagonal matrix and the lower block diagonal matrix 0 has dimension $m \times m$ (recall that $A \in \Re^{m \times n}$). We refer to (2.4) as the augmented system.

Eliminating Δx from (2.4) we get $ADA^t \, \Delta y = r_p + A(Dr_d - Z^{-1} r_a)$, which is called the *normal equations* system.

We remark that the augmented and normal systems of equations for problems with bounded variables have the same structure as the systems of equations in the standard form. Therefore, the ideas developed here can be readily applied to such problems.

2.2. Approaches for Solving Linear Systems of Equations

Using the Cholesky factorization of the normal systems of equations for computing the search directions in interior point methods is by far the most widely used approach (see for example (Adler et al.,1989; Czyzyk et al., 1999; Gondzio, 2012). However, the factored matrix can have much less sparsity and is often more ill-conditioned than the matrix of the augmented system (2.4). Solving the augmented system by direct methods is another option (Bergamaschi et al., 2004; Al-Jeiroudi et al., 2008). However, the sequence of pivots in the decomposition depends on the numerical values and this approach while robust, is in general more expensive (Gondzio, 2012). Moreover, the direct approach cannot be applied for some classes of large scale problems due to memory and/or time limitations. For these problems, a preconditioned iterative method for the solution of the linear system would be the chosen approach (Bergamaschi et al., 2007; Bocanegra et al., 2007; Oliveira & Sorensen, 2005).

In most applications, however, it is essential to modify a linear system of equations that is very difficult to solve to obtain an equivalent system that is easier to solve by the iterative method. This technique is known as preconditioning. Again, whenever the choice between the augmented systems or the normal systems of equations arises, most researchers chose to solve the augmented system since it is less ill-conditioned (Bergamaschi et al., 2007; Chai & Toh, 2007; Gondzio, 2012). We work with the normal systems equations because it is positive definite, allowing the use of the conjugate gradient method. In addition, the splitting preconditioner works very well in the final iterations where the linear system is highly ill-conditioned.

Since the iterative methods require matrix only for computing matrix-vector products, there is no need to compute the normal equations unless the preconditioner depends on it. On the other hand, a new trend in the past few years is to use of simple linear programming methods in order to give an advanced starting point for interior point methods. This reduces the total number of iterations. The von Neumann's algorithm is one of the first to be used in such applications since its iteration is very cheap and it has fast initial convergence (Dantzig & Thapa, 1992).

In this work, we perform a few iterations of the optimal adjustment algorithm for *p* coordinates (Ghidini et al., 2012), a simple linear programming method, before the change of preconditioners, to deliver a point closer to an optimal solution for the splitting preconditioner. This approach closes the gap in the transition of preconditioners for some tested problems.

2.3. The Splitting Preconditioner

The Splitting preconditioner, proposed in Oliveira and Sorensen (2005), is a generalization of the preconditioner proposed in Resende and Veiga (1993) in the context of the minimum cost network flow problem.

The splitting preconditioner is computed as follows:

Let $A = [B\ N\ ]P$, where $P \in \Re^{n\times n}$ is a permutation matrix such that $B \in \Re^{m\times m}$ is nonsingular and $N \in \Re^{m\times(n-m)}$, then

$$ADA^t = \begin{bmatrix} B & N \end{bmatrix} PDP^t \begin{bmatrix} B^t \\ N^t \end{bmatrix} = \begin{bmatrix} B & N \end{bmatrix} \cdot \begin{bmatrix} D_B & 0 \\ 0 & D_N \end{bmatrix} \cdot \begin{bmatrix} B^t \\ N^t \end{bmatrix} = BD_B B^t + ND_N N^t .$$

The preconditioner is given by $D_B^{-1/2} B^{-1}$ and the preconditioner matrix M is as follows:

$$M = D_B^{-1/2} B^{-1} \left(ADA^t\right) B^{-t} D_B^{-1/2} = I_m + GG^t ,$$

where $G = D_B^{-1/2} B^{-1} N D_N^{1/2}$.

The product $B^{-1}N$ can be seen as a scaling of the linear programming problem. Close to a solution, at least $n - m$ entries of D are small. Thus, with a suitable choice of the B columns, the diagonal entries of D_B^{-1} and D_N become very small. In this situation, G approaches the zero matrix, M approaches the identity matrix and both the largest eigenvalue of M and $\kappa_2(M)$ approach one.

2.4. Controlled Cholesky Factorization Preconditioner

The Controlled Cholesky Factorization (CCF) preconditioner, designed for solving general positive definite systems (Campos & Birkett, 1998), can be seen as a variation of the incomplete Cholesky factorization. The main objective of this factorization is to build a preconditioned matrix that has grouped eigenvalues and which is near the identity in order to accelerate the convergence of the conjugate gradient method.

The Cholesky factorization of the matrix ADA^t is as follows:

$$ADA^t = LL^t = \bar{L}\bar{L}^t + R,$$

where L represents the factor obtained when the factorization is complete, $\bar{L}$ represents a factor obtained when factorization is incomplete and R is a remainder matrix.

Matrix $\bar{L}$ is used as a preconditioner matrix for ADA^t,

$$\bar{L}^{-1}ADA^t\bar{L}^{-t} = \left(\bar{L}^{-1}L\right)\left(L^t\bar{L}^{-t}\right) = \left(\bar{L}^{-1}L\right)\left(\bar{L}^{-t}L\right)^t .$$

Let $F = L - \bar{L}$. Replacing L in the last equation, we have

$$\bar{L}^{-1}\left(ADA^t\right)\bar{L}^{-t} = \left(I_m + \bar{L}^{-1}F\right)\left(I_m + \bar{L}^{-1}F\right)^t.$$

Note that when $\bar{L} \approx L$ then $F \to 0$ and, therefore, $\bar{L}^{-1}\left(ADA^t\right)\bar{L}^{-t} \to I_m$.

The controlled Cholesky factorization is based on the minimization of the Frobenius norm of F. Therefore, when $||F|| \to 0$ then $||R|| \to 0$.

Consider the following problem:

$$\text{Min } \|F\|_F^2 = \sum_{j=1}^m c_j \text{ , with } c_j = \sum_{j=1}^m \left|l_{ij} - \bar{l}_{ij}\right|^2 ,$$

where, l_{ij} are elements of L.

Splitting c_j in two sums leads to:

$$c_j = \sum_{k=1}^{t_j+\eta}\left|l_{i_k j} - \bar{l}_{i_k j}\right|^2 + \sum_{k=t_j+\eta+1}^{m}\left|l_{i_k j}\right|^2 ,$$

where t_j is the number of nonzero entries below the diagonal in the jth column of matrix ADA^t and η is the number of extra entries allowed per column in the incomplete factorization.

The first summation contains all $tj + \eta$ nonzero entries of the jth column of $\bar{L}$. The second one has only the remaining entries of the complete factor L

which do not have the corresponding entries in $\bar{L}$. Thus, the problem can be solved using the following heuristic:

1. Increasing η (allowing more fill-ins). The term c_j should decrease because the first summation contains more elements.
2. Choosing the $tj + \eta$ largest entries of $\bar{L}$ in an absolute value for fixed η. In this case, the largest entries are in the first summation leaving only the smallest l_{ij} in the second one.

The preconditioner $\bar{L}$ is built by columns. Consequently, it needs only the jth column of ADA^t at each time, avoiding the computation of the normal equations system.

2.5. A Hybrid Preconditioner

Matrix D changes significantly from one interior point iteration to the next and it becomes highly ill-conditioned in the final ones. For this reason, it is difficult to find a preconditioning strategy that has a good performance over the entire course of the interior point iterations.

In Bocanegra et al. (2007) it was proposed to apply the conjugate gradient method to solve the normal equations system preconditioned by a hybrid preconditioner matrix M,

$$M^{-1}ADA^tM^{-t}\bar{y} = M^{-1}\left(AD\left(r_d - X^{-1}r_a\right) + r_p\right),$$

where $\bar{y} = M^t\Delta y$. This approach assumes the existence of two phases during interior point iterations. In the first one, the controlled Cholesky preconditioner is used to build matrix M. After the change of phases, matrix M is built using the splitting preconditioner.

In Velazco et al. (2010), a heuristic for change of preconditioners was proposed. If the number of iterations needed for the conjugate gradient method to achieve convergence is greater than $m/6$, the parameter η in the controlled Cholesky factorization is increased, i e, $\eta = \eta + 10$. The change occurs when η exceeds a fixed maximum η. However, this approach can fail to achieve convergence for some classes of linear programming problems when the controlled Cholesky factorization is not longer effective and at the same time, the splitting preconditioner is not yet ready for the job. In order to improve this approach, simple algorithms are used in the iteration where the change of phases occurs giving a more advanced point towards optimality just before the splitting preconditioner is applied (Ghidini et al., 2012).

3. Practical Aspects

In this section we discuss a few issues concerning the splitting preconditioner not directly related to finding the linearly independent set of columns that form *B*.

3.1. Inexact Solutions

An idea that immediately comes to mind when using iterative procedures for solving the linear systems is to relax the required tolerance. Thus, we start the interior point method with a relaxed tolerance (10^{-4}) and, whenever an iteration does not (at least) halve the gap ($x^t z$), the tolerance is changed to the square root of machine epsilon.

In the context of the predictor-corrector variant there is another place for applying this idea. Recall that for computing the search directions, two linear systems are solved. The first one gives the perturbation parameter and the nonlinear correction for the Newton's method. The second one can be written in such a way that it gives already the final search directions. Thus, the first linear system may be solved with a more relaxed tolerance than the second one.

3.2. Discarding Dependent Rows

In order for the splitting preconditioner to be built, the constraint matrix *A* cannot have dependent rows. The following procedure finds the dependent rows and discards them before the interior point method starts.

The techniques to be described in the next section for finding *B* can be applied to the columns ordered by degree. Moreover, rows containing entries that are part of singleton columns can be ignored in this factorization since these rows are necessarily independent. This idea can be applied in the resulting matrix until there are no longer any singleton columns. Thus, finding dependent rows is inexpensive most of the time. Actually, there are problems like those with only inequality constraints where no factorization is performed at all. This factorization can be computed even more efficiently (Andersen, 1995).

4. Computing the Splitting Preconditioner

This class of preconditioners is not a competitive alternative to the direct method approach for computing the Cholesky factorization without a careful implementation. This is due to the computation of an *LU* factorization where the set of independent columns is unknown at the start of the factorization. This factorization may be too expensive for two reasons. First, it may generate too many fill-in entries. Second, it may be necessary to factor too many columns before the completion of the factorization since the dependent columns must be discarded. Several techniques for the implementation of a competitive code are discussed below. Most of the techniques presented here concern the computation of the *LU* factorization and are used on the numerical experiments presented in Section 5.

Given a good choice of columns from *A* to form *B*, this preconditioner should work better close to a solution, where the linear systems are highly ill-conditioned. A strategy to form *B* is to minimize $\left\| D_B^{1/2} B^{-1} N D_N^{-1/2} \right\|$. This problem is hard to solve. However, it can be approximately solved with a simple heuristic. Select the first *m* linearly independent columns of AD^{-1} with smallest 2-norm. This choice of columns tends to produce better conditioned matrices as the interior point method approaches a solution, where the linear systems are highly ill-conditioned (Velazco et al., 2010).

A partition of matrix *A* has been used before as a preconditioner for network interior point methods (Resende & Veiga, 1993). In this situation *B* is a minimum spanning tree and is easy to find. Therefore, the splitting preconditioner can be viewed as a generalization. We also remark that the rules for choosing the set of columns are not the same.

4.1. Scaling the Columns

Looking at the expression $G = D_N^{-1/2} N^t B^{-t} D_B^{1/2}$ again, it is tempting to scale the matrix after selecting the columns of *B* such that $\| G \| \approx 0$. Lemma 1 shows that this idea is not easy to implement since the scaling will disappear on the preconditioned matrix (Oliveira, 1997).

Lemma 1 *Consider the following scaling of the augmented system*

$$\begin{bmatrix} C & 0 \\ 0 & I \end{bmatrix} \cdot \begin{bmatrix} -D & A^t \\ A & 0 \end{bmatrix} \cdot \begin{bmatrix} C & 0 \\ 0 & I \end{bmatrix} \cdot \begin{bmatrix} C^{-1} & 0 \\ 0 & I \end{bmatrix} \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} C & 0 \\ 0 & I \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

Then the preconditioned matrix is independent of the scaling matrix C.

Proof. The scaled matrix is given by

$$\begin{bmatrix} -CDC & \tilde{A}^t \\ \tilde{A} & 0 \end{bmatrix},$$

where $\tilde{A} = AC$. Its preconditioned form is as follows

$$P^t \begin{bmatrix} I & D_B^{1/2} B^{-1} N D_N^{-1/2} \\ D_N^{-1/2} N^t B^{-t} D_B^{1/2} & -I \end{bmatrix} P,$$

where $AP^t = [B\ N]$. This form is independent of C.

It is still possible to use the idea of scaling with good results. After computing the first *LU* factorization, the columns not on *B* can be rescaled giving a new linear programming problem. With a proper choice of the scaling factor, the first linear systems will be better conditioned than before and yet the problem will not be badly scaled. Before a second *LU* factorization is computed, the original linear programming problem is recovered by undoing the scaling. Therefore, the linear programming problem will be properly scaled for the remainder of the procedure and the first iterations of the interior point method generate better conditioned systems which are the most difficult for this class of preconditioners.

4.2. Keeping the Set of Columns

A good property of the splitting preconditioner is that it enables us to work with a selected set of columns for some iterations. As a consequence, the preconditioner is very cheap to compute for these iterations.

It is important to note that keeping the matrix B from previous iterations does not mean the same preconditioner will be kept since D will change from iteration to iteration. Thus, this strategy gives different preconditioners at each iteration that are very easy to compute. However, such preconditioners do not have the best column set after the first factorization according to the heuristic.

Table 1: KEN13 - New Factorization versus Keeping *LU*

Version	**Factorizations**	**M-Flops**	**Factorization LU time (s)**	**Triangular systems time (s)**
Standard	25	117	74.16	24.82
Keep *LU*	4	426	15.46	56.06

Table 1 illustrates this idea. In column version we have the standard approach of computing a factorization at every iteration against the idea for keeping the factorization. A new *LU* factorization is computed whenever the preconditioned conjugate gradient method takes more than $n/30$ iterations to converge, where n is the dimension of the linear system. Notice that this version computes only four factorizations and it is faster than the standard approach. However, it takes more floating point operations on average for solving a linear system. The explanation is that there is a high overhead on computing the *LU* factorization. In contrast, the solution of the triangular systems is straightforward with practically no overhead.

In the experiments given later we change the set whenever the iterative method takes more iterations than a certain threshold value ($m^{1/2}$) or when the solution given by it is not accurate.

4.3. Incomplete LU Factorization

It was observed in practice that the *LU* factorization often generates too many fill-in entries. The reason is that no reordering procedure for reducing the number of fill-ins can be used since the columns of the matrix are not known until they are accepted as linearly independent.

Here we discuss another possibility. It consists of computing an incomplete *LU* factorization. The standard incomplete factorization, where the nonzero structure of the original matrix coincides with the nonzero structure of the triangular matrices *L* and *U*, does not work well for this problem (Oliveira, 1997). On the other hand, the use of drop tolerance seems to be a viable approach. The idea is to eliminate any entry smaller than a preset value. For a carefully chosen tolerance, this technique can be very useful and it actually gives better performance on some of the problems tested in very preliminary experiments. This line of research deserves more investigation.

4.4. Using Indicators

Another approach that can be exploited for reducing work on the factorizations when the interior point method is close to the solution is the use of indicators (El-Bakry et al., 1994). An indicator is a tool for determining if a column is not part of any optimal basis before the method converges. Such columns can thus be eliminated from the problem. In the context of this work, indicators can be used to keep these columns at the end of the list for finding an independent set of columns, saving work on the factorization. Since these columns are not being eliminated from the problem, it is possible to be less rigorous on the way they are determined without taking the risk of getting a wrong solution for the linear programming problem. Observe that the diagonal entries of *D* are also valid indicators. Thus, the approach adopted as standard in Section 5 actually uses a kind of indicator for reordering the columns although on a different manner compared to the one described here.

4.5. LU Factorization

For this application, the most economic way to compute the *LU* factorization is to work with the delayed update form. When a linearly dependent column appears, it is eliminated from the factorization and the method proceeds with the next column in the ordering.

One of the main drawbacks of a straightforward implementation of the splitting preconditioner is the excessive fill-in in the *LU* factorization. A good technique consists of interrupting the factorization when excessive fill-in occurs and reordering the independent columns found thus far by the number of nonzero entries. The factorization is then restarted from scratch and the process is repeated until *m* independent columns are found. In our implementation we consider a factorization to have excessive fill-in if it produces more nonzero entries than the normal equations system.

4.6. Avoiding Dependent Columns

A more sophisticated approach should identify a set of columns that have dependent relationship with another column. Information about such relationship can be used to reduce computational time and effort on the following factorizations by not considering these columns whenever they appear behind this set in the new ordering.

In order to have an efficient search for these sets and to avoid excessive use of computer memory, this type of information can be stored at the bit level. Thus, if A has n columns, a set can be stored on n bits and operating with these bits will be much faster than managing arrays of indices. Moreover, memory restriction can be a critical issue if arrays are used for storing these sets for large scale problems. A hash function could also be used to speed up the implementation of this approach.

4.7. Computing a Second LU Factorization

A second factorization is applied on the chosen set of independent columns using standard techniques for computing an efficient sparse LU factorization. This approach improves the results significantly for some problems because the reduction of the floating point operations on the iterative linear system solver compensates the extra work for computing the factorization. It also benefits better form the predictor-corrector variant because the preconditioner is used for solving two linear systems. Therefore, the second factorization is always performed. As a welcome side effect, it is not necessary to store U in the factorizations that determine B.

Table 2 illustrates this savings for problem Truss. This problem is part of the Netlib test collection of linear programming problems (Gay, 1985). The dimension of the linear system is 1000. Only the iterations where the second LU factorization is computed are shown.

The second factorization is computed whenever the number of nonzero entries of L plus U is more than four times the dimension of the linear system. The second approach saves work because the time for solving the linear systems with a more sparse preconditioner compensates the time for computing the second factorization.

The following techniques are the default options for the second LU factorization on our code. We stress that it is not possible to use them on the first LU factorization because the structure of B is not known prior to the factorization.

The columns are permuted by the ordering of B^tB given by the minimum degree ordering. We included a threshold parameter for the choice of the pivot. At each step of the factorization, we chose a row permutation with the pivot being chosen among all candidates within the threshold. The one with least entries on its row for the remaining columns of the original matrix is chosen. We also find strongly connected components to rewrite the matrix in a block triangular form, as usually done in square *LU* factorization.

Table 2: Truss number of nonzero entries

IP	**Nonzero Entries**		**Factorization time (s)**		**Triangular systems time (s)**	
Iteration	**First LU**	**Second LU**	**First LU**	**Second LU**	**First LU**	**Second LU**
7	21713	15202	0.32	0.35	1.47	0.92
8	27367	15797	0.23	0.25	1.50	0.83
9	28159	19195	0.19	0.19	1.13	0.52
10	37436	17933	0.26	0.28	1.01	0.44
11	41139	18051	0.35	0.37	1.33	0.32
12	37369	18140	0.50	0.53	2.22	0.25
13	37060	17540	0.71	0.75	0.55	0.22
14	41858	17616	0.46	0.48	0.81	0.20
15	39400	20661	0.52	0.54	0.83	0.19
16	40832	17191	0.35	0.38	1.40	0.17
17	43442	21716	0.37	0.40	2.11	0.18
18	40826	23510	0.37	0.40	0.38	0.16
M-Flops	**106.8**	**86.1**				

4.8. Early Detection of Dependent Columns

One difficulty in determining the subset of independent columns relates to the number of dependent columns visited in the process. An approach is to verify whether a column is dependent or not during the delayed update form of the *LU* factorization. If we find that a candidate column is already dependent on the first say, *k* columns, it is useless to continue updating the candidate column for the remaining columns of *L*.

4.9. Symbolically Dependent Columns

Given a column ordering, we want to find the first set of *m* independent columns. The brute force approach for this problem consists in computing the factorization column by column and discarding the (nearby) dependent columns along the way. The strategies developed here will indicate when a column can be ignored in the factorization. The set of independent columns found by these techniques is the same set obtained by the brute force approach.

Symbolically dependent columns are columns that are linearly dependent in structure for all numerical values of their nonzero entries. The idea is to find a set of say *k* columns with nonzero entries in at most *k*-1 rows. This set of columns is symbolically dependent.

Let us first consider a square matrix for simplicity. In this situation, the problem is equivalent to permuting nonzero entries onto the diagonal. This problem is equivalent to finding a matching of a bipartite graph where the rows and columns form the set of vertices and the edges that are represented by the nonzero entries. This idea was first used by Duff (1981) and it is applied as a first step for permuting a matrix to block triangular form. If a nonzero entry cannot be assigned to the diagonal in the matching process for a given column, that column is symbolically dependent.

In Coleman & Pothen (1987) this idea is extended to rectangular matrices. The authors are concerned with finding a set of independent columns of the matrix which gives a sparse *LU* factorization. Thus, the columns are reordered by degree and the matching algorithm applied giving a set of candidate columns, denoted here as *key columns*, which are not symbolically dependent.

Our idea for using the key columns comes from the fact that the number of independent columns before the *k*th key column on the matrix is at most *k*-1. Therefore, it is possible to speed up the *LU* factorization whenever we find *k*-1 numerically independent columns located before the *k*th key column. The speed up is achieved by skipping all the columns from the current one to the *k*th key column.

4.10. Matching During the Factorization

Sometimes the use of key columns does not save too much work. The reason is that often these columns are numerically dependent. Another way to save floating point operations is to compute the matching during the factorization. Thus, before we update the column we verify whether it is symbolically dependent or not. If it is, the column is discarded and the factorization continues with the next column.

This technique can save computational work because the matching can be done on the original matrix instead of the factored one. Moreover, no floating point operation is performed. If many columns are dependent, the overhead caused for the ones that are not dependent is compensated reducing the overall time for computing the factorization.

4.11. Symbolically Independent Columns

Symbolically independent columns are columns that are linearly independent in structure for all numerical values of their nonzero entries. A powerful strategy consists in moving the symbolically independent columns to the beginning of the ordered list since those columns are necessarily going to be in the factorization. Then these columns can be reordered further in order to reduce the number of fill-ins in the *LU* factorization. Notice that the symbolically dependent columns can be ignored in this step. Thus, we are concerned only with the key columns given by the matching algorithm.

We are not aware of any efficient algorithm for finding all the symbolically independent columns from a given ordered set. Therefore, we use heuristic approaches to identify some of the symbolically dependent columns.

On the description of the heuristic below, we say that column *j* is the first entry column of row *i* if *j* contains the first nonzero entry in row *i* on the ordered set. We consider a column *j* symbolically independent given an ordered set if at least one of the following rules applies:

1. Column *j* is the first entry column of at least one row;
2. Column *j* is the second entry column of a row *i* and the first entry column of row *i* is also first entry column for at least another row not present on column *j*.

This set of rules guarantees that the columns selected are symbolically independent but it does not guarantee that all symbolically independent columns are found.

4.12. Key Columns and Independent Columns

Another use for key columns is to anticipate the sparse structure of the *B* matrix to be factored. This information can be used to reduce the number of fill-in entries on the factorization. This idea works fine for some problems but it deteriorates the preconditioner computational performance on other too much. Therefore, it cannot be used as the default approach. One criterion to decide on using this approach is the number of symbolically independent columns found. If this number is close to the total number of columns, the key columns give a better approximation of the sparsity pattern of *B* since most of the columns in the factorization are known.

4.13. Merging Symbolically Independent and Dependent Columns

After reordering the symbolically independent columns, it may be possible to reduce still further the number of fill-ins in the factorization by merging the symbolically independent and dependent list of columns using the number of nonzero entries as the criterion. This is allowed whenever the symbolically independent columns remain so.

It is very expensive to verify whether the columns remain symbolically independent at every step of the merging process. Therefore, we use the first ordering of the columns as a cheap heuristic. Thus, we move up on the list a symbolically dependent column with lower degree provided it remains behind the symbolically independent columns with lower index on the first ordering. This idea can be implemented very efficiently.

By placing columns with lower degree into the front we hope to reduce the number of fill-ins in the factorization. However, since the symbolically dependent columns are less likely to be in the factorization, this approach is not as effective as other approaches presented in this section.

4.14. Strongly Connected Components

This approach is applied to the key columns. Since the key columns are determined by a matching procedure, a permutation for computing the strongly connected components is already at hand. Given the strongly connected components, their columns are reordered by the splitting criterion.

In Oliveira & Sorensen (2005) it has been proved that we can look for the first symbolically dependent column in its own component considering only the rows from the respective component. All columns with smaller index in the ordering are symbolically independent.

Notice that with this approach, the heuristics for finding symbolically independent columns can be applied inside each diagonal block.

5. Numerical Experiments

In this section we present several numerical experiments with the hybrid preconditioner approach. The experiments are meant to show how the techniques for computing the splitting preconditioner work together and to determine which ones are going to be adopted as default.

The procedures for solving the linear systems with the splitting preconditioner are coded in C and applied within the PCx code (Czyzyk et al., 1999), a state of the art interior point methods implementation. PCx's default parameters are used except that multiple corrections are not allowed and all tolerances for the interior point are set to 10^{-8}.

All the experiments are carried out on an Intel Core 2 Duo 64 bits, 2GB RAM and 2.2GHz with operating system Linux. The floating point arithmetic is IEEE standard double precision.

5.1. Stopping Criterion

The preconditioned conjugate gradient method is used with a termination criterion set by the Euclidean residual norm $\| \, . \, \|_2$. For solving both systems (affine direction and final direction), the termination criteria is set as $\|r_k\| < 10^{-4}$. When the optimality gap is less than 10^{-5} or change of phases is detected, the criteria change to $\|r_k\| < 10^{-8}$. The maximum number of iterations of the conjugate gradient method is equal to the system dimension.

5.2. Test Problems

In this work, 41 test problems were considered, all they are freely available. The problems are from NETLIB (Gay, 1985) (http://www.netlib.org) and QAPLIB (Burkard et al., 1991) collections.

Table 3 contains the basic statistics about the test problems. Column **Dimension** gives the number of rows and columns of the test problems after preprocessing.

Table 3: Problems statistics

Problem	Dimension	Collection	Problem	Dimension	Collection
25fv47	788 × 1843	NETLIB	forplan	121 × 447	NETLIB
adlittle	55 × 137	NETLIB	hil12	1355 × 3114	QAPLIB
agg2	514 × 750	NETLIB	israel	174 × 316	NETLIB
agg3	514 × 750	NETLIB	kb2	43 × 68	NETLIB
bandm	240 × 395	NETLIB	maros	655 × 1437	NETLIB
blend	71 × 111	NETLIB	nug05	210 × 225	QAPLIB
bnl2	1964 × 4008	NETLIB	nug06	372 × 486	QAPLIB
boeing1	331 × 697	NETLIB	nug07	602 × 931	QAPLIB
boeing2	125 × 264	NETLIB	nug08	912 × 1632	QAPLIB
bore3d	81 × 138	NETLIB	nug12	3192 × 8856	QAPLIB
capri	241 × 436	NETLIB	nug15	6330 × 22275	QAPLIB
chr20b	4219 × 7810	QAPLIB	qap12	2794 × 8856	QAPLIB
chr20c	4219 × 7810	QAPLIB	qap15	5698 × 22275	QAPLIB
chr22b	5587 × 10417	QAPLIB	rou10	839 × 1765	QAPLIB
chr25a	8149 × 15325	QAPLIB	rou20	7359 × 37640	QAPLIB
d6cube	403 × 5444	NETLIB	scr12	1151 × 2784	QAPLIB
degen2	444 × 757	NETLIB	scr15	2234 × 6210	QAPLIB
degen3	1503 × 2604	NETLIB	scr20	5079 × 15980	QAPLIB
e226	198 × 429	NETLIB	ste36b	27683 × 131076	QAPLIB
els19	4350 × 13186	QAPLIB	stocfor2	1980 × 2868	NETLIB
finnis	438 × 935	NETLIB			

5.3. Obtained Results

In order to compare the various techniques, we have adopted as a standard version the one which considers the preconditioner hybrid approach and uses the techniques described in sections 3.2, 4.2, 4.4, 4.7, 4.9, 4.11, 4.12 but not the techniques in sections 4.3 and 4.10.

Table 4 presents a comparison of the standard version total running time against versions that consider only one non default technique in computing the splitting preconditioner. The remaining columns have the following meanings:

- **Split:** only the splitting preconditioner is used;
- **NoRefac:** matrix B is not refactored;
- **B:** matrix B is computed at each iteration;
- **NoSing:** singleton rows and columns are not searched;
- **NoKey:** does not use key columns;
- **Match:** matching during LU;
- **IncLU:** compute incomplete LU;

- **NoLI:** does not reorder linearly independent columns;
- **NoOrd:** does not reorder columns during *LU* factorization.

A comparison between columns **Split** and **Stand** leads to the conclusion that starting with the splitting preconditioner affects the performance approach negatively in the great majority of cases. Moreover, some problems, not presented here, are solved before the change of phase occurs. For such problems, the time difference in favor of the standard approach is even lager. Finally, the standard approach is more robust since it solves a larger number of problems.

Using key columns produces mixed results. Using it achieve better results for 43% of the problems while it worsens the performance in 34% of the problems. With respect to the matching, (Sec 4.10), these values are 42% and 27%, respectively. However, for most the larger problems, it seems to be advisable not to perform the matching. The incomplete *LU* factorization does not change the time in a significant way. However, it loses robustness, in particular among larger problems.

The standard approach that reorders the linearly independent columns is both faster and more stable than where the reordering is not done. Not reordering during the *LU* factorization also has a negative effect on the performance of the standard performance approach.

The approach that does not refactor the matrix is slower than the standard one, especially for large-scale problems. For instance, the ste36b problem running time increases about 66%. This result is confirmed by the nonzero entries average number which significantly reduces in the refactored matrix. For ste36b problem, this reduction goes up to 72% (see Table 5). A similar result with respect to running time is achieved when matrix *B* is computed at each iteration, in particular for large-scale problems, as well. When singleton rows and columns are not searched, many problems are not solved and the remaining ones reveal a time increase of order about 6.2 in average.

Table 5 compares the average number of nonzero entries in the standard refactored matrix approach with the approach that does not refactor the matrix for problems with large total running time. Furthermore, in order to give a better idea of the problems size, the **A** column gives the constraint matrix number of entries of the preprocessed problems. The $\mathbf{ADA^t}$ column shows the number of nonzeros entries of the lower triangular half and **L** column has the number of nonzero entries of the Cholesky factor, which is not needed in the presented approach.

Regarding iteration counts the results are almost the same for all variations studied and are not shown. In majority of the cases, there was no better approach. In some cases the difference was just one iteration.

Finally, we can conclude that the standard approach is more robust and efficient in solving all the tested problems. However, there is room for improvement. If we select the best time for all the other variants to compare against the standard approach, the latter is faster for about 32% of the tested problems.

Table 4: Total running time (s) - * means that the method failed

Problem	**Stand**	**Split**	**NoRefa**	**B**	**NoSing**	**NoKey**	**Match**	**IncLU**	**NoLI**	**NoOrd**
25fv47	2,66	8,23	2,69	2,67	*	2,67	2,65	2,66	2,64	2,69
adlittle	0,00	0,01	0,00	0,01	*	0,01	0,01	0,01	0,00	0,00
agg2	0,67	0,80	0,67	0,66	1,47	0,67	0,66	0,67	0,70	0,68
agg3	0,50	0,65	0,50	0,45	1,18	0,50	0,49	0,49	0,49	0,50
bandm	0,15	0,31	0,16	0,15	*	0,16	0,16	0,16	0,16	0,16
blend	0,00	0,01	0,01	0,01	0,00	0,01	0,01	0,01	0,00	0,00
bnl2	7,32	11,20	7,35	6,13	*	7,38	7,23	7,30	7,38	7,41
boeing1	0,27	0,23	0,28	0,23	*	0,27	0,27	0,27	0,28	0,28
boeing2	0,03	0,03	0,03	0,04	*	0,03	0,03	0,03	0,03	0,03
bore3d	0,02	0,02	0,02	0,02	0,06	0,02	0,02	0,02	0,02	0,03
capri	0,08	0,11	0,09	0,09	*	0,09	0,09	0,09	0,09	0,09
chr20b	15,86	10,04	16,27	14,36	16,15	15,58	16,04	15,91	16,15	16,02
chr20c	12,19	6,78	12,28	12,26	12,74	12,18	12,22	12,24	12,70	12,31
chr22b	16,35	18,04	16,25	16,12	17,90	15,76	16,22	16,35	16,52	17,73
chr25a	43,26	44,62	43,14	43,72	46,12	41,32	43,55	43,37	43,73	48,01
d6cube	2,02	3,73	2,03	2,05	*	2,09	1,93	2,17	1,94	2,04
degen2	0,37	0,32	0,35	0,43	*	0,29	0,35	0,35	0,33	0,36
degen3	8,35	6,54	8,44	13,53	*	5,56	8,32	8,34	7,55	8,50
e226	0,19	0,36	0,18	0,18	*	0,18	0,18	0,18	0,18	0,18
els19	93,80	112,37	94,98	336,61	267,18	58,59	97,39	94,20	123,13	95,19
finnis	0,26	*	0,27	0,19	*	0,26	0,26	0,26	*	0,26
forplan	0,17	0,75	0,17	0,17	0,18	0,17	0,17	0,16	0,17	0,17
hil12	7,65	11,60	7,61	9,54	8,27	6,53	7,65	7,68	8,89	7,65
israel	0,13	0,12	0,13	0,13	*	0,14	0,12	0,13	0,13	0,13
kb2	0,00	0,00	0,00	0,01	0,00	0,00	0,00	0,00	0,00	0,00
maros	2,50	*	2,56	2,49	*	2,51	2,48	2,49	2,82	2,54
nug05	0,03	0,03	0,02	*	0,05	0,03	0,03	0,02	*	0,03
nug06	0,12	0,12	0,12	0,12	0,15	0,11	0,12	0,12	0,12	0,12
nug07	0,48	0,62	0,49	0,58	0,48	0,41	0,48	0,48	0,53	0,50
nug08	1,18	1,76	1,20	1,50	1,25	1,31	1,20	1,17	1,37	1,20
nug12	156,10	313,59	169,31	184,11	147,76	189,64	155,09	*	*	188,36
nug15	1592,32	5585,4	2156,89	1900,5	*	1669,4	1581,81	*	2011,0	1610,35
qap12	151,83	265,37	176,75	189,50	144,55	139,42	151,26	*	168,44	153,12
qap15	4711,82	*	6617,60	4897,6	*	3859,0	*	*	3562,1	*
rou10	1,74	2,63	1,75	2,15	1,80	1,61	1,77	1,75	2,27	1,75
rou20	1769,07	8199,0	1375,91	10557,	1776,06	1523,4	1766,24	*	2074,5	1798,90
scr12	1,80	1,89	1,81	1,88	8,19	1,74	1,80	1,80	1,86	1,80
scr15	11,62	12,84	11,66	20,75	*	8,56	11,49	11,56	13,61	11,69
scr20	138,56	169,12	138,91	224,79	858,26	92,98	144,43	137,95	169,11	139,76
ste36b	20957,7	*	34725,2	34768,	*	*	25477,8	*	38422,	25633,6
stocfor2	2,56	*	2,63	2,11	*	2,57	2,53	2,55	2,59	2,60

Table 5: Number of nonzero entries

Problem	A	ADA^t	L	Stand	NoRefac
nug12	33528	57217	2793152	400587	576890
nug15	85470	150448	11053969	950503	1468514
qap12	33528	60181	2138580	382126	556144
qap15	85470	155986	8197968	998538	2443669
rou20	152980	356689	20818131	3211137	3175678
ste36b	512640	1564487	176625274	4278712	14838724

6. Conclusions

An important advantage of the splitting preconditioner is that it becomes better in some cases as the interior point method advances towards an optimal solution since the linear systems are difficult to solve by iterative methods using traditional preconditioners. That is a very interesting characteristic given that the linear systems are known to be very ill-conditioned close to a solution. However, an efficient implementation of the splitting preconditioner is not trivial. In this work, we have presented numerical experiments that illustrate the performance of several strategies that speed up the computation of the splitting preconditioner. A standard method is proposed and approaches that can lead to future improvement are suggested. These include scaling the normal equations system, computing an incomplete LU factorization, using a hash function to quickly detect dependent columns and develop new ways for fast detect symbolically independent columns.

On the other hand, a generic preconditioner should be used in the first interior point method iterations when the linear systems are not much ill-conditioned and the nice features of the splitting preconditioners are not yet at work. At the transition stage, simple algorithms for linear programming problems can be used in order to add robustness to the proposed approach.

Acknowledgements

This research was sponsored by the Brazilian Council for the Development of Science and Technology (CNPq) and Foundation for the Support of Research of the State of São Paulo, (FAPESP).

References

Adler, I., Resende, M. G. C., Veiga G. & Karmarkar, N. (1989). An implementation of Karmarkar's algorithm for linear programming. *Math. Programming,* 44, 297-335.

Al-Jeiroudi, G., Gondzio, J. & Hall, J. A. J. (2008). Preconditioning indefinite systems in interior point methods for large scale linear optimization. *Optimization Methods and Software,* 23, 345-363.

Andersen, E. D. (1995). Finding all linearly dependent rows in large-scale linear programming. *Optimization Methods and Software*, 6, 219-227.

Bergamaschi, L., Gondzio J. & Zilli G. (2004). Preconditioning indefinite systems in interior point methods for optimization. *Computational Optimization and Applications*, 28, 149-171.

Bergamaschi, L., Gondzio, J., Venturin, M. & Zilli, G. (2007). Inexact constraint preconditioners for linear systems arising in interior point methods. *Computational Optimization and Applications*, 36, 137-147.

Bocanegra, S., Campos, F. F. & Oliveira, A. R. L. (2007). Using a hybrid preconditioner for solving large-scale linear systems arising from interior point methods. *Computational Optimization and Applications*, 36, 149-164.

Burkard, R. E., Karish, S. E. & Rendl, F. (1991). QAPLIB - A quadratic assignment problem library problems. *European Journal of Operational Research*, 55, 115-119.

Campos, F. F. & Birkett, N. R. C. (1998). An efficient solver for multi-right hand side linear systems based on the CCCG(η) method with applications to implicit time-dependent partial differential equations. *SIAM J. Sci. Comput.*, 19, 126-138.

Chai, J. S. & Toh, K. C. (2007). Preconditioning and iterative solution of symmetric indefinite linear systems arising from interior point methods for linear programming. *Computational Optimization and Applications*, 36, 221-247.

Coleman, T. F. & Pothen, A. (1987). The null space problem II. Algorithms. *SIAM J. Alg. Disc. Meth.*, 8, 544-563.

Czyzyk, J., Mehrotra, S., Wagner, M. & Wright, S. J. (1999). PCx an interior point code for linear programming. *Optimization Methods & Software*, 11(2), 397-430.

Dantzig, G. B. & Thapa, M. N. (1992). *Linear Programming 2: Theory and Extensions*. Springer, New York.

Duff, I., Erisman, A. & Reid, J. (1986). *Direct methods for sparse matrices*. Clarendon Press, Oxford.

Duff, I. S. (1981). On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Software*, 7, 315-330.

El-Bakry, A. S., Tapia, R. A. & Zhang, Y. (1994). A study of indicators for identifying zero variables in interior-point methods. *SIAM Rev.*, 36, 45-72.

Gay, D. M. (1985). Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter*, 13, 10-2.

Ghidini, C. T. L. S., Oliveira, A. R. L., Silva, J. & Velazco, M. I. (2012). Combining a hybrid preconditioner and an optimal adjustment algorithm to accelerate the convergence of interior point methods. *Linear Algebra and its Applications*, 218, 1267-1284.

Gondzio, J. (1996). Multiple centrality corrections in a primal-dual method for linear programming. *Computational Optimization and Applications*, 6, 137-156.

Gondzio, J. (2012). Interior point methods 25 years later. *European Journal of Operational Research*, 218, 587-601.

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

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

Momoh, J. A., El-Hawary, M. E. & Adapa, R. (1999). A review of selected optimal power flow literature to 1993, part II Newton, linear programming and interior point methods. *IEEE Transactions on Power Systems*, 14, 105-111.

Oliveira, A. R. L. (1997). *A new class of preconditioners for large-scale linear systems from interior point methods for linear programming,* tech. rep., PhD Thesis, TR97-11, Department of Computational and Applied Mathematics, Rice University, Houston TX.

Oliveira, A. R. L. & Sorensen, D. C. (2005). A new class of preconditioners for large-scale linear systems from interior point methods for linear programming. *Linear Algebra and Its Applications*, 394, 1-24.

Resende, M. G. C. & Veiga, G. (1993). An efficient implementation of a network interior point method. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 12, 299-348.

Velazco, M. I., Oliveira, A. R. L. & Campos, F. F. (2010). A note on hybrid preconditions for large scale normal equations arising from interior-point methods. *Optimization Methods and Software,* 25, 321-332.