SIAM J. MATRIX ANAL. APPL.
Vol. 13, No. 1, pp. 191–201, January 1992

A BLOCK-*LU* UPDATE FOR LARGE-SCALE LINEAR PROGRAMMING*

SAMUEL K. ELDERSVELD† AND MICHAEL A. SAUNDERS†

Dedicated to Gene Golub on the occasion of his 60th birthday

Abstract. Stable and efficient updates to the basis matrix factors are vital to the simplex method. The "best" updating method depends on the machine in use and how the update is implemented. For example, the classical product-form update can take advantage of the vector hardware on current supercomputers, and this helps compensate for its well-known drawbacks. Conversely, the method of Bartels and Golub performs well on conventional machines, but is difficult to vectorize.

With vectorization in mind, we examine a method based on the block-*LU* factors of an expanding basis. The partitioned matrix involved was introduced by Bisschop and Meeraus [*Math. Programming*, 13 (1977), pp. 241–254], [*Math. Programming*, 18 (1980), pp. 7–15]. The update itself was proposed by Gill, Murray, Saunders, and Wright [*SIAM J. Sci. Statist. Comput.*, 5 (1984), pp. 562–589].

The main advantages of the block-*LU* update are that it is stable, it vectorizes well, and compared to the product-form update, the nonzeros increase at about two-thirds the rate. The update has been incorporated into MINOS and tested on 30 large, sparse linear programming problems. Results are given from runs on a Cray Y-MP.

Key words. matrix factorization, updating, simplex method, linear programming

AMS(MOS) subject classifications. 65F05, 65F50, 65K05, 90C05

1. Introduction.

We wish to use the simplex method [Dan63] to solve the standard linear programming (LP) problem,

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & l \leq x \leq u, \end{aligned}$$

where A is an m by n matrix and c, x, l, u, and b are of appropriate dimension.

The simplex method is an *active-set method* for optimization. At each iteration a rank-one modification (in the form of a column update) is made to a basis matrix B associated with constraints active at the current point. After k updates, the columns of A may be permuted to the form $(B_k \quad N_k)$. The next update replaces the pth column a_r of B_k by a column a_q from N_k. It can be written

$$B_{k+1} = B_k + (a_q - a_r)e_p^T, \tag{1}$$

where e_p is the pth column of the identity matrix. The basis is used to solve for the search direction y and the dual variables π in the following linear systems:

$$B_k y = a_q, \tag{2}$$

$$B_k^T \pi = c_k, \tag{3}$$

where c_k contains the objective coefficients corresponding to the columns of B_k.

* Received by the editors January 11, 1991; accepted for publication (in revised form) July 19, 1991. This research was supported in part by National Science Foundation grant ECS-8715153 and Office of Naval Research grant N00014-90-J-1242. The first author's research was also supported by an IBM Graduate Technical Fellowship.

† Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305-4022 (na.eldersveld@na-net.ornl.gov and mike@sol-michael.stanford.edu).

Stable and efficient basis updates are vital to the computational success of the simplex method. The "best" updating method depends on the machine in use and how the update is implemented. For example, the classical *product-form* (PF) update,

$$B_k = B_0 T_1 T_2 \cdots T_k, \tag{4}$$

can take advantage of the vector hardware on current supercomputers such as the Cray X-MP and Y-MP. This helps compensate for its potential instability and for the typically high rate of growth of nonzeros in the "eta" vectors representing the elementary triangular factors T_k.

Conversely, the *Bartels–Golub* (BG) update [Bar71],

$$B_k = L_k U_k, \quad L_k = L_0 T_1 \cdots T_k, \tag{5}$$

performs well on conventional machines [Rei82], [GMSW87] but is difficult to vectorize fully because each T_k may be a product of triangular factors involving short vectors, and U_k is altered in an unpredictable manner. The *Forrest–Tomlin* (FT) update [FT72], also described by (5), makes simpler changes to L_k and U_k and is probably more amenable to vectorization.

With vector machines in mind, we examine two further updates in §§2 and 3. We then discuss implementation details for the second method and present computational results comparing a *block-LU* method to the BG update.

2. The Schur-complement update. As an alternative to (2), Bisschop and Meeraus [BM77], [BM80] drew attention to an augmented system $\tilde{B}_k \tilde{y} = \tilde{a}_q$ of the form

$$\begin{pmatrix} B_0 & V_k \\ U_k & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} a_q \\ 0 \end{pmatrix}, \tag{6}$$

where

$$V_k = (a_{q_1} \ \cdots \ a_{q_k}), \qquad U_k = (e_{p_1} \ \cdots \ e_{p_k})^T. \tag{7}$$

Initially, B_0 is defined as a basis matrix at the start of the first iteration. After a number of iterations it may be necessary to factorize the current basis B_k and redefine it to be B_0. Each a_{q_j} $(j = 1, \cdots, k)$ corresponds to a basic column from A that has become basic since the last refactorization of B_0.

System (6) is equivalent to (2). To see this, note that the equation $U_k y_1 = 0$ sets k elements of y_1 to zero, so that the remaining elements of y_1 when combined with y_2 give the solution $y \in \Re^m$. Specifically, y may be formed by setting $y \leftarrow y_1$ and overwriting $y(j) \leftarrow y_2(i)$ for $i = 1, \cdots, k$, where j is defined as the unit-vector index of the ith row of U_k.

The solution to (6) can be found by solving in order

$$B_0 w = a_q, \tag{8}$$

$$C_k y_2 = U_k w, \tag{9}$$

$$B_0 y_1 = a_q - V_k y_2, \tag{10}$$

where $C_k = U_k B_0^{-1} V_k$ is the *Schur-complement* matrix. In general, this method requires two solves with B_0 as well as a single solve with the matrix C_k, which will have a maximal dimension of k. If a_q happens to be a column originally from B_0, we

have $a_q = B_0 e_s$ for some s. In this case $w = e_s$ and (9) reduces to $C_k y_2 = e_i$, where the ith row of U_k is e_s^T. In addition, (10) can be written as $B_0(y_1 - e_s) = -V_k y_2$, so that a_q itself need not be known.

Likewise, the solution to (3) can be found by solving the equivalent system

$$\begin{pmatrix} B_0^T & U_k^T \\ V_k^T & \end{pmatrix} \begin{pmatrix} \pi_1 \\ \pi_2 \end{pmatrix} = \begin{pmatrix} c_0 \\ d_k \end{pmatrix} \tag{11}$$

and taking $\pi = \pi_1$. That is, by solving in order

$$B_0^T z = c_0, \tag{12}$$

$$C_k^T \pi_2 = V_k^T z - d_k, \tag{13}$$

$$B_0^T \pi_1 = c_0 - U_k^T \pi_2. \tag{14}$$

During Phase 1 of the simplex method, c_0 in (11) and (12) may change for each k, but in Phase 2, system (12) need be solved only once each time the basis is refactorized. In addition, from (14) we see that $U_k^T \pi_2 = c_0 - B_0^T \pi_1$. This implies that π_2 corresponds to the set of reduced costs for columns of B_0 that are currently nonbasic.

Note that for most updates, a new column is added to V_k to obtain V_{k+1}. However, the updates occasionally involve replacing or deleting columns of V_k. From now on, k refers to the number of columns in V_k, not the simplex iteration number. The matrices involved then have the following dimensions: B_k is $m \times m$, N_k is $m \times (n-m)$, V_k is $m \times k$, U_k is $k \times m$, and C_k is $k \times k$.

2.1. Advantages. The Schur-complement (SC) update for linear programming was first described by Bisschop and Meeraus [BM77], [BM80], one of whose aims was to provide an updating technique with storage requirements that are *independent of the problem size m*. This is a unique feature.

The SC update shares an important advantage with the PF update, in that the factors L_0 and U_0 are used many times without modification. On a vector machine, the triangular solves with these factors can therefore be reorganized to take advantage of the vector hardware, as recently shown in [ER90]. The greater stability of the SC update allows the overhead associated with this reorganization to be spread over 100 iterations (say), whereas the PF update may fail a stability test at any stage (in the worst case after only one or two iterations).

A further advantage of the SC and PF updates is that it is only necessary to solve systems with B_0 and B_0^T; we do not need to access the columns of B_0 for pricing. This may be important for specially structured problems. See [GMSW84, pp. 578–580] for further discussion.

2.2. Stability. The matrix in (6) has the following block-triangular factorization:

$$\tilde{B}_k = \begin{pmatrix} B_0 & V_k \\ U_k & \end{pmatrix} = \begin{pmatrix} I & \\ U_k B_0^{-1} & I \end{pmatrix} \begin{pmatrix} B_0 & V_k \\ & -C_k \end{pmatrix}. \tag{15}$$

Recalling that U_k is composed of unit vectors, we see that if B_0 is "reasonably well conditioned," then the first triangular factor is also reasonably well conditioned. In such cases, the Schur complement C_k tends to reflect the condition of $\tilde{B}_k$, which is essentially the same as the condition of the true basis B_k.

This means that when C_k is updated, ill-conditioning need not persist (because certain rows and columns of C_k are explicitly added or deleted). For example, suppose

bases $B_0, B_1, \cdots, B_k$ are all well conditioned *except* for B_j. Then all of the Schur complements will be well conditioned except C_j, and hence all of the basis factorizations will be well conditioned except for the jth. This property, shared by the BG update, defines our meaning of stability.

In short, the SC update is essentially as stable as the BG update, *provided* B_0 *is well conditioned.* This cannot be said of the PF or FT updates. (Of course, the BG update remains superior in being stable regardless of the condition of B_0.)

2.3. Comments. A discussion of the Schur complement may be found in [Cot74]. Implementations of a Schur-complement method for general LP problems are described in [Pro85], and for specially structured linear programs in [Eld88].

The original descriptions of Bisschop and Meeraus [BM77], [BM80] involved updating C_k^{-1} explicitly (not a stable process). Proctor [Pro85] presented two implementations, one updating C_k^{-1} as a dense matrix, the other maintaining sparse LU factors of C_k. The latter is to be preferred for stability reasons. Since k can be limited to 100 (say), we believe it is more efficient to maintain dense factors of C_k; see §4.1.

Our original aim was to investigate the performance of the SC update on general LP problems. The method was implemented, but it soon became evident that the additional solves with B_0 and B_0^T were excessively expensive compared to the BG update. The following variation was therefore chosen as a means of trading workspace for time.

3. A block-*LU* update. Rather than using (15) we may factorize $\tilde{B}_k$ in the following manner:

$$\tilde{B}_k = \begin{pmatrix} B_0 & V_k \\ U_k & \end{pmatrix} = \begin{pmatrix} B_0 & \\ U_k & -C_k \end{pmatrix} \begin{pmatrix} I & Y_k \\ & I \end{pmatrix}, \tag{16}$$

where

$$B_0 Y_k = V_k, \qquad C_k = U_k Y_k. \tag{17}$$

We see that the solution to (6) and hence $B_k y = a_q$ may be obtained from

$$B_0 w = a_q, \tag{18}$$

$$C_k y_2 = U_k w, \tag{19}$$

$$y_1 = w - Y_k y_2. \tag{20}$$

Likewise, the solution to $B_k^T \pi = c_k$ may be obtained from

$$C_k^T \pi_2 = Y_k^T c_0 - d_k, \tag{21}$$

$$B_0^T \pi_1 = c_0 - U_k^T \pi_2. \tag{22}$$

The block-*LU* update was first discussed in [GMSW84].[1] All updating information is carried along via the Schur-complement matrix C_k and the matrix of transformed columns Y_k. The updates to these matrices will be discussed in the next section. Note that C_k is composed of some of the rows of Y_k. It may be described as "some of the rows and columns of the simplex tableau associated with the starting basis B_0."

[1] It was termed a *stabilized product-form* update because the columns of Y_k are handled similarly to the "eta" vectors in the classical product-form update, and because the factors of B_0 are not altered. Note, however, that (16) is an explicit block-triangular factorization. Nothing is held in product form.

3.1. Advantages. The block-*LU* update has most of the advantages of the SC update, in terms of using B_0 as a "black box." The storage for C_k remains independent of m. By storing Y_k we reduce the work per iteration of the simplex method by a solve with B_0 and (in Phase 1) a solve with B_0^T. For many iterations when a row of Y_k is needed to update C_k, we avoid a further solve with B_0^T.

Comparing the right-hand sides of (10) and (20), we see that the term $V_k y_2$ has become $Y_k y_2$, which is usually somewhat more expensive. The analogous term $Y_k^T c_0$ in (21) costs little because most of it does not require updating.

3.2. Stability. The block-*LU* update possesses the same stability properties as the SC update. The main requirement again is that B_0 be reasonably well conditioned.

In practice we can prevent excessive ill-conditioning in B_0 by replacing certain columns with the unit vectors associated with slack variables, according to the size of the diagonal elements in the initial *LU* factors. A rather lax tolerance is needed to prevent altering the basis after every factorization and thereby impeding convergence of the simplex method. In the computational tests reported here, provision was made to altered B_0 if its condition appeared to be greater than $\epsilon^{-2/3} \approx 10^{10}$ (where the machine precision was $\epsilon \approx 10^{-15}$). However, no such alterations occurred. Thus, after every 100 iterations the current B_k was always accepted as B_0, and no numerical difficulties were encountered.

4. Implementation issues. For the block-*LU* update to be efficient, we must be able to update C_k and Y_k efficiently at each iteration. The updates to these matrices consist of four cases:

1. Add a row and column to C_k, and add a column to Y_k.
2. Replace a column of C_k and Y_k.
3. Replace a row of C_k, leaving Y_k unchanged.
4. Delete a row and column of C_k, and delete a column from Y_k.

Each of these cases depends on the type of column entering or leaving the basis and whether or not the columns were in the initial B_0. A description of each case follows.

Case 1. The entering column is from N_0, and the leaving column is from B_0. A row and column are added to C_k:

$$U_{k+1} = \begin{pmatrix} U_k \\ e_p^T \end{pmatrix} \quad \text{and} \quad Y_{k+1} = \begin{pmatrix} Y_k & w \end{pmatrix}, \tag{23}$$

$$C_{k+1} = U_{k+1}\, B_0^{-1}\, V_{k+1} = \begin{pmatrix} C_k & U_k w \\ e_p^T Y_k & \delta \end{pmatrix}, \tag{24}$$

where $B_0 w = a_q$ and $\delta = e_p^T w$. Note that w is already available from (8) in the simplex algorithm. It becomes a new column of Y_k.

Case 2. The entering column is from N_0 and the leaving column is from V_k (not from B_0). A column of C_k is again replaced by $U_k w$, which is already available from the simplex algorithm. The dimension of C_k stays the same. A column in Y_k is replaced by the new transformed column w.

Case 3. The entering column is from B_0 and the leaving column is from B_0. A row of C_k is replaced with the pth row of Y_k. The dimension of C_k stays the same. Y_k is not altered.

Case 4. The entering column is from B_0 and the leaving column is from V_k (and not from B_0). We delete a row and column from C_k and we delete the corresponding column from Y_k.

4.1. Storage of C_k. The size of C_k will never be larger than the *refactorization frequency.* Since this is relatively small for most large-scale LP problems (we used 100), it is efficient to treat C_k as a dense matrix.

For maximum reliability, we maintain a dense orthogonal factorization $Q_k C_k = R_k$, where Q_k is orthogonal and R_k is upper triangular. The techniques for updating the QR factors of C_k involve sweeps of *plane rotations* as discussed in [GGMS74]. A set of routines called QRMOD were used for this purpose. For slightly greater efficiency, Q_k and R_k may be updated using sweeps of *stabilized elimination matrices;* see [Cli77].

4.2. Storage of Y_k. As Y_k has a row dimension of m, the method of dealing with this matrix is important. Y_k consists of transformed columns that have entered the basis since the last refactorization. We must be able to do matrix-vector multiplies with Y_k (20) and Y_k^T (21) as well as fetch rows of Y_k (24). The sparsity of each column of Y_k depends on the sparsity of the basis itself as well as the sparsity of each of the entering basic columns.

Since the use of indirect addressing reduces performance on most vector computers, indirect adressing should be avoided for all except very sparse vectors. On the other hand, performing computations with vectors containing a very large proportion of zero elements is also inefficient. With this in mind, each column of Y_k is stored in one of two ways depending on its density. We have used the following dynamic storage scheme for Y_k:

1. A column of Y_k that has a density of at least `NTHRSH` is considered to be *dense.* Such columns are stored "as is" and not packed. In the computational tests, a value of `NTHRSH` $= 0.40$ was used.
2. Columns with density less than `NTHRSH` are considered *sparse* and are packed in a conventional column list. For each column, the nonzero elements of these vectors are stored contiguously, along with a parallel array of row indices, the number of nonzeros, and a pointer to the first nonzero.

The average sparsity for Y_k's columns for each of the test problems is given in Table 4. A row of Y_k can be extracted trivially from columns in dense form. Packed columns require a search for the desired row index, which can usually be vectorized.

Dense columns of Y_k are stored separately from the sparse columns in order to make operations with the dense columns vectorizable. Thus, the storage array for Y_k consists of a dense part and a sparse part. The updates to Y_k consist of adding, deleting and replacing columns. Each case is described below.

1. When a new column is added to Y_k, the column is simply appended to the end of the "dense" or "sparse" arrays for Y_k. Dense columns are stored "as is" and sparse columns are packed in a conventional column list.
2. For simplicity, column deletion was implemented by moving all later columns one place to the left (thereby overwriting the deleted column and recovering its storage). The operations are essentially the same whether the deleted column is dense or sparse. Half of Y_k must be moved on average, but the copy operation is vectorizable and cheap. Also, less than half of the updates require deletion.
3. Column replacement occurs in one of two ways. When both new and old columns are dense, the new column simply overwrites the old column. In all other cases, the old column is deleted from Y_k (2 above) and the new column is appended to Y_k (1 above).

TABLE 1
Problem specifications.

No.	Problem	Rows	Cols	Elem	Objective value
1	80bau3b	2263	2266	29063	9.8722822814E+05
2	bp822	822	825	11127	5.5018458595E+03
3	cycle	1904	1907	21322	-5.2263930249E+00
4	czprob	930	933	14173	2.1851966988E+06
5	etamacro	401	404	2489	-7.5571519542E+02
6	fffff800	525	528	6235	5.5567961167E+05
7	ganges	1310	1313	7021	-1.0958627396E+05
8	greenbea	2393	2396	31499	-7.2462397960E+07
9	grow22	441	444	8318	-1.6083433648E+08
10	nesm	663	666	13988	1.4076079892E+07
11	perold	626	629	6026	-9.3807558690E+03
12	pilot.ja	941	944	14706	-6.1131579663E+03
13	pilot.we	723	726	9218	-2.7201045880E+06
14	pilot4	411	414	5145	-2.5811392641E+03
15	pilotnov	976	979	13129	-4.4972761882E+03
16	pilots	1442	3652	43220	-5.5760732709E+02
17	scfxm2	661	664	5229	3.6660261565E+04
18	scfxm3	991	994	7846	5.4901254550E+04
19	scrs8	491	494	4029	9.0429998619E+02
20	scsd6	148	151	5666	5.0500000078E+01
21	scsd8	398	401	11334	9.0499999993E+02
22	sctap3	1491	1494	17554	1.4240000000E+03
23	ship08l	779	782	17085	1.9090552114E+06
24	ship12l	1152	1155	21597	1.4701879193E+06
25	ship12s	1152	1155	10941	1.4892361344E+06
26	stair	357	360	3857	-2.5126695119E+02
27	stocfor2	2158	2161	9492	-3.9024408538E+04
28	tdesg1	3500	4050	18041	4.3560773922E+04
29	tdesg5	4215	22613	105002	4.3407357993E+04
30	woodw	1099	1102	37478	1.3044763331E+00

5. Computational results. In this section we compare numerical results obtained from an implementation of the algorithm described in §3. The standard basis update in MINOS 5.3 [MS87] is the Bartels–Golub update. For a complete discussion of LUSOL, the package of basis routines in MINOS 5.3, see [GMSW87].

The implementation of the block-*LU* update has been included as an option in a specially modified version of MINOS 5.3. The new version, MINOS/SC 5.3, includes other options including a special pricing routine designed especially for vector computers described in [FT88], and a vectorization algorithm for the solution of triangular systems of equations described in [ER90]. These options were disabled for the present computational tests.

The purpose of the tests is to demonstrate the efficiency of the new update and show that for vector machines the method is more efficient than the Bartels–Golub update on a representative set of large, sparse problems. The two algorithms are labeled BG for the Bartels–Golub update and BLU for the block-*LU* update. The tests consist of comparing timings of BG and BLU by solving 30 linear programming test problems. Many of these problems are available from the *netlib* collection [Gay85]. The test problem specifications are given in Table 1. The smallest *netlib* test problems were omitted from the results, as some timing categories for these problems were less than 1/100th of a second on the machine used.

TABLE 2
Update results.

	Method	BG:	$\bar{U}_{BG}$	BLU:	$\bar{U}_{BLU}$	$\bar{C}$	$\bar{U}_{BG}/\bar{U}_{BLU}$
No.	Problem name	Total update time (sec)	Mean update time (μsec)	Total update time (sec)	Mean update time (μsec)	Mean size C_k	Update speed-up
1	80bau3b	35.79	30137.88	3.87	3385.17	33.99	8.90
2	bp822	13.50	20079.65	3.83	5471.75	25.66	3.67
3	cycle	11.60	36633.00	2.91	9745.26	37.74	3.76
4	czprob	2.49	16289.44	0.67	4228.94	37.20	3.85
5	etamacro	0.39	7130.00	0.50	8427.22	34.73	0.85
6	fffff800	0.97	10281.49	0.79	7898.30	36.01	1.30
7	ganges	1.22	17342.51	0.44	6180.37	40.44	2.81
8	greenbea	118.99	46059.83	18.87	7391.08	31.06	6.23
9	grow22	1.28	18483.14	0.81	11508.06	40.50	1.61
10	nesm	3.64	12021.19	1.57	5621.24	30.99	2.14
11	perold	6.64	17083.67	1.82	4787.23	23.68	3.57
12	pilot.ja	14.98	23820.10	3.31	5134.65	24.27	4.64
13	pilot.we	7.55	15853.32	2.18	4737.04	24.65	3.35
14	pilot4	1.76	12291.56	0.66	4567.76	23.23	2.69
15	pilotnov	6.78	24923.93	1.57	5679.31	26.80	4.39
16	pilots	117.55	72976.57	8.89	5388.31	24.12	13.54
17	scfxm2	0.88	11478.61	0.64	8299.71	39.82	1.38
18	scfxm3	1.97	16756.42	0.98	8236.08	40.40	2.03
19	scrs8	0.64	9984.70	0.34	6497.42	29.36	1.54
20	scsd6	0.40	3611.14	0.85	7396.47	33.84	0.49
21	scsd8	2.83	8415.58	3.58	10152.39	40.66	0.83
22	sctap3	2.61	26277.24	0.64	5952.07	41.13	4.41
23	ship08l	0.71	13613.07	0.14	2673.06	42.42	5.09
24	ship12l	2.10	18806.31	0.31	2747.94	40.92	6.84
25	ship12s	0.93	17370.72	0.20	3632.95	42.79	4.78
26	stair	0.67	12352.67	0.29	5270.27	24.60	2.34
27	stocfor2	7.29	36568.73	2.19	9562.93	39.75	3.82
28	tdesg1	24.98	60392.59	2.31	5950.65	39.59	10.15
29	tdesg5	271.51	80266.15	27.61	7774.24	40.12	10.32
30	woodw	7.67	20271.52	2.63	6807.28	37.37	2.98
	MEAN	22.34	23919.09	3.18	6370.17	34.26	4.14

5.1. Test environment. The computational tests were performed on an 8-processor Cray Y-MP supercomputer. Only one processor was used. The operating system was UNICOS version 5.1, and the MINOS code was compiled using the CFT77 compiler with full optimization. Each run was made as a batch job.

For each test the number of iterations and total solution time are recorded in Table 4. The solution time was measured by timing the MINOS subroutine `M5SOLV`. The options used for MINOS were the standard MINOS/SC options, namely `PARTIAL PRICE 10, SCALE OPTION 2, FACTORIZATION FREQUENCY 100`. The set of problems was then run with (BLU) and without (BG) the `SCHUR-COMPLEMENT` option.

For purposes of evaluating the block-LU update, the following items were deemed to be of interest for each method:

1. Total and average time spent updating the basis.
2. Total time spent solving for dual variables π and the search direction y using the basis factors.
3. Average solve times with the basis factors.

TABLE 3
Solve results.

	Method	BG:		BLU:	
No.	Problem name	Mean solve π (μsec)	Mean solve y (μsec)	Mean solve π (μsec)	Mean solve y (μsec)
1	80bau3b	40568.12	32337.06	41853.63	27891.02
2	bp822	28050.47	27790.85	26730.54	22025.21
3	cycle	34750.42	44877.62	37200.98	39155.63
4	czprob	17566.18	12823.63	19565.78	10986.37
5	etamacro	8594.14	8356.12	10980.39	6787.33
6	fffff800	10509.23	12726.54	12594.22	11379.08
7	ganges	14002.71	17897.94	17465.92	16229.53
8	greenbea	67136.65	54699.46	70031.12	47843.61
9	grow22	21071.95	22581.79	19877.60	16715.92
10	nesm	14057.41	14433.41	14532.32	10878.63
11	perold	22935.79	23822.10	20340.22	16273.83
12	pilot.ja	30307.37	32987.48	26348.98	23197.10
13	pilot.we	24727.20	25234.13	22966.24	18282.15
14	pilot4	18750.67	18488.82	13718.79	10770.45
15	pilotnov	29968.91	32639.40	26490.74	23707.26
16	pilots	66454.81	65155.80	48638.38	44065.14
17	scfxm2	13580.72	13348.56	16225.99	11929.60
18	scfxm3	19622.53	19358.95	22415.36	17280.10
19	scrs8	12217.33	11776.33	12735.71	7971.36
20	scsd6	6011.64	4875.61	8348.29	4777.12
21	scsd8	16193.41	12426.66	18668.06	11411.83
22	sctap3	14512.53	20346.93	18591.30	19643.66
23	ship08l	14065.56	9978.80	17703.06	10714.05
24	ship12l	16048.90	12260.41	18865.25	12051.95
25	ship12s	16948.28	12683.65	20275.87	12718.61
26	stair	17300.74	17629.60	12552.57	9661.55
27	stocfor2	30451.35	40716.96	34382.52	36011.83
28	tdesg1	52146.37	50159.20	54627.08	46676.17
29	tdesg5	80847.35	69401.81	85958.77	67876.06
30	woodw	27375.91	24530.58	28535.34	20750.70
	MEAN	26225.82	25544.87	26640.70	21188.76

5.2. Updates. Time spent updating the basis was measured by timing the appropriate portion of the MINOS subroutine `M5SOLV`. The total and average updating times are recorded in Table 2. These results dramatize the efficiency of the block-*LU* update for the Cray Y-MP. In 27 of the 30 test problems the BLU method gave faster mean and total updating times than BG. The average update speedup was 4.14. A point of interest is that while update times grew for the larger problems using method BG, the average update time remained fairly constant for method BLU. The average BG update time ranged from 3611–80266 microseconds, while the range was 2673–11508 microseconds for the BLU update.

5.3. Solves. The average solve times for the two methods are quite similar, as exhibited in Table 3. It is important to note that although it was not performed here, the solves with L_0 *and* U_0 can be vectorized with method BLU. The solves with L_0 may be vectorized for method BG but as U_k is updated explicitly with this method,

TABLE 4
Overall Problem Results.

	Method	BG:		BLU:		
No.	Problem name	Itns	Soln. time (secs)	Itns	Soln. time (secs)	Mean dens. Y_k
1	80bau3b	11963	206.22	11425	166.94	.022
2	bp822	6792	71.70	6999	58.64	.621
3	cycle	3198	51.92	2987	39.63	NA
4	czprob	1544	11.97	1595	10.48	.016
5	etamacro	550	2.00	594	2.25	.159
6	fffff800	953	4.85	996	4.95	.277
7	ganges	708	5.78	718	5.34	.051
8	greenbea	26094	622.14	25527	493.56	.223
9	grow22	704	6.39	703	4.99	.691
10	nesm	3058	21.96	2792	17.66	.121
11	perold	3923	35.74	3801	25.71	NA
12	pilot.ja	6350	80.46	6445	58.94	.640
13	pilot.we	4805	48.17	4611	37.24	.719
14	pilot4	1446	9.98	1446	6.90	.577
15	pilotnov	2747	35.04	2773	26.65	.568
16	pilots	16267	577.24	16494	347.56	.736
17	scfxm2	772	4.33	772	4.24	.094
18	scfxm3	1184	9.61	1184	8.83	.104
19	scrs8	647	3.30	521	2.33	.155
20	scsd6	1127	3.12	1153	3.76	.343
21	scsd8	3400	22.17	3531	24.45	.338
22	sctap3	1003	9.72	1070	8.80	.024
23	ship08l	526	4.02	523	3.66	.011
24	ship12l	1125	12.10	1113	10.58	.007
25	ship12s	538	4.60	544	4.12	.006
26	stair	551	3.62	551	2.45	.655
27	stocfor2	2014	31.50	2292	30.28	.088
28	tdesg1	4177	95.50	3878	69.27	NA
29	tdesg5	34177	1334.49	35518	1144.78	.070
30	woodw	3822	65.17	3860	58.56	NA
	MEAN	4872.16	113.16	4880.53	89.45	.291

U_0 is not constant between refactorizations. This means that it is possible to decrease solution times with the factors of B_0 using method BLU even further.

5.4. Comparison with the product-form update. On average, the density of the columns of Y_k will be similar to that of the eta vectors in the classical product-form update. Note, however, that over a period of 100 iterations the average number of columns in Y_k is only 25 to 40, with a mean of 34. This means that the number of transformed vectors used in solving systems of equations is lower for the block-*LU* method than for the PF update, where the average would be 50 if stability requirements allow 100 updates. Since the size of the additional matrix C_k is small on average (25 to 40), this suggests that the block-*LU* update requires fewer floating-point operations per solve as well as lower storage requirments than the PF update on large problems. The ratio is $34/50 \approx 2/3$.

6. Conclusions. 1. A block-*LU* update technique is a viable alternative to a standard (Bartels–Golub) updating technique when vectorization is available.

2. Numerical experiments running a modified version of MINOS 5.3 on a Cray Y-MP showed the block-*LU* update to be superior to Bartels–Golub updating on 27 of 30 test problems.

3. Average solve times with basis factors using the block-*LU* update were comparable to the solve times using the standard method.

4. Use of the block-*LU* update reduced CPU times by approximately 21 percent on these test problems. Vectorization of all the solves with L_0, U_0, L_0^T, U_0^T as in [ER90] would give a further marked improvement.

Acknowledgments. The authors are indebted to Bill Kamp and John Gregory of Cray Research, Inc. for encouraging the research and providing computer time for the computational tests, and to the referee for some helpful comments.

REFERENCES

[Bar71] R. H. BARTELS, *A stabilization of the simplex method*, Numer. Math., 16 (1971), pp. 414–434.

[BM77] J. BISSCHOP AND A. MEERAUS, *Matrix augmentation and partitioning in the updating of the basis inverse*, Math. Programming, 13 (1977), pp. 241–254.

[BM80] ———, *Matrix augmentation and structure preservation in linearly constrained control problems*, Math. Programming, 18 (1980), pp. 7–15.

[Cli77] A. K. CLINE, *Two subroutine packages for the efficient updating of matrix factorizations*, Report TR-68, Department of Computer Sciences, The University of Texas, Austin, TX, 1977.

[Cot74] R. W. COTTLE, *Manifestations of the Schur-complement*, Linear Algebra Appl., 8 (1974), pp. 189–211.

[Dan63] G. B. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[Eld88] S. K. ELDERSVELD, *Stochastic linear programming with discrete recourse: An algorithm based on decomposition and the Schur-complement update*, Engineer Thesis, Department of Operations Research, Stanford University, Stanford, CA, 1988.

[Eld89] ———, MINOS/SC *User's guide supplement*, Internal Tech. Report, Department of Industry, Science and Technology, Cray Research, Inc., Mendota Heights, MN, 1989.

[ER90] S. K. ELDERSVELD AND M. C. RINARD, *A vectorization algorithm for the solution of large, sparse triangular systems of equations*, Report SOL 90-1, Department of Operations Research, Stanford University, Stanford, CA, 1990.

[FT72] J. J. H. FORREST AND J. A. TOMLIN, *Updating triangular factors of the basis to maintain sparsity in the product-form simplex method*, Math. Programming, 2 (1972), pp. 263–278.

[FT88] ———, *Vector processing in simplex and interior methods for linear programming*, IBM Res. Report No. RJ 6390 (62372), IBM, Yorktown Heights, NY, 1988.

[Gay85] D. M. GAY, *Electronic mail distribution of linear programming test problems*, Mathematical Programming Society COAL Newsletter, 13 (1985), pp. 10–12.

[GGMS74] P. E. GILL, G. H. GOLUB, W. MURRAY, AND M. A. SAUNDERS, *Methods for modifying matrix factorizations*, Math. Comput., 4 (1974), pp. 505–535.

[GMSW84] P. E. GILL, W. MURRAY, M. A. SAUNDERS, AND M. H. WRIGHT, *Sparse matrix methods in optimization*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 562–589.

[GMSW87] ———, *Maintaining LU factors of a general sparse matrix*, Linear Algebra Appl., 88/89 (1987), pp. 239–270.

[MS87] B. A. MURTAGH AND M. A. SAUNDERS, MINOS 5.1 *user's guide*, Report SOL 83-20R, Department of Operations Research, Stanford University, Stanford, CA, 1987.

[Pro85] P. E. PROCTOR, *Implementation of the double-basis simplex method for the general linear programming problem*, SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 567–575.

[Rei82] J. K. REID, *A sparsity exploiting variant of the Bartels–Golub decomposition for linear programming bases*, Math. Programming, 24 (1982), pp. 55–69.