Electronic Transactions on Numerical Analysis.
Volume 26, pp. 161-177, 2007.

ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

ON THE EFFICIENT UPDATE OF RECTANGULAR LU-FACTORIZATIONS SUBJECT TO LOW RANK MODIFICATIONS*

PETER STANGE†, ANDREAS GRIEWANK‡, AND MATTHIAS BOLLHÖFER†

Abstract. In this paper we introduce a new method for the computation of KKT matrices that arise from solving constrained, nonlinear optimization problems. This method requires updating of null-space factorizations after a low rank modification. The update procedure has the advantage that it is significantly cheaper than a re-factorization of the system at each new iterate. This paper focuses on the cheap update of a rectangular LU-decomposition after a rank-1 modification. Two different procedures for updating the LU-factorization are presented in detail and compared regarding their costs of computation and their stability. Moreover we will introduce an extension of these algorithms which further improves the computation time. This turns out to be an excellent alternative to algorithms based on orthogonal transformations.

Key words. KKT-system, LU-factorization, low-rank modification, quasi-Newton method

AMS subject classifications. 15A23, 65F05, 65F30, 65K05, 90C53

1. Introduction. This work is motivated by the solution of the following constrained optimization problem:

$$\min_{x\in\mathbb{R}^n} f(x) \qquad \text{subject to} \qquad \begin{cases} c_i(x) = 0 & i \in \mathcal{E} \\ c_j(x) \le 0 & j \in \mathcal{I} \end{cases} \quad \text{where } \mathcal{I}\cap\mathcal{E} = \emptyset,\ \ (\mathcal{I}\cup\mathcal{E}) = \{1,...,k\}.$$

If the number of $m \le k$ active constraints is known, the optima of this problem are locally characterized as saddle points of the Lagrange function

$$\mathcal{L}(x,\lambda) = f(x) + \lambda^T c(x) = f(x) + \sum_{i=1}^{m} \lambda_i c_i(x), \quad \text{where } \lambda = (\lambda_1, ..., \lambda_m)^T.$$

These saddle points can be computed by solving the stationary condition

$$0 = \nabla_{x,\lambda}\mathcal{L}(x,\lambda) \equiv [g(x,\lambda), c(x)],$$

where $g(x,\lambda) = \nabla f + \sum_{i=1}^{m} \lambda_i \nabla c_i(x)$. This can be done by the quasi-Newton method introduced in [7]. Thereby a sequence of linearized KKT systems of the form

$$\begin{bmatrix} B_k & A_k^T \\ A_k & 0 \end{bmatrix} \begin{bmatrix} s_k \\ \sigma_k \end{bmatrix} = - \begin{bmatrix} g_k \\ c_k \end{bmatrix} \tag{1.1}$$

has to be solved. Here the matrices $A_k \in \mathbb{R}^{m\times n}$ and $B_k \in \mathbb{R}^{n\times n}$ (with $n \ge m$) approximate the Jacobian $\nabla c(x)^T$ of the active constraints and the Hessian of the Lagrange function $\nabla_x^2\mathcal{L} = \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 c_i(x)$. The vectors s_k and σ_k are the

*Received February 7, 2006. Accepted for publication September 4, 2006. Recommended by Y. Saad. This work was supported by the DFG-research center MATHEON "Mathematics for key technologies" in Berlin.
†Institute for Mathematics, Technische Universität Berlin, Germany. (p.stange@tu-bs.de).
‡Institute for Mathematics, Humboldt-Universität zu Berlin, Germany (griewank@mathematik.hu-berlin.de, bolle@tu-bs.de).

ETNA
Kent State University
etna@mcs.kent.edu

current optimization steps from which the new point and the new Lagrange multipliers can be obtained by

$$x_{k+1} = x_k + s_k \quad \text{and} \quad \lambda_{k+1} = \lambda_k + \sigma_k.$$

This is a full step. The convergence can be globalized by reduced steps

$$x_{k+1} = x_k + \alpha_k s_k \quad \text{and} \quad \lambda_{k+1} = \lambda_k + \sigma_k.$$

with a step length α_k, e.g., obtained by a line search. Throughout the paper we make no assumptions on the sparsity pattern of ∇c and $\nabla^2 \mathcal{L}$, i.e., A_k, B_k might be dense matrices. The approximate Jacobian A_k and the approximate Hessian B_k will be updated in every step of the optimization procedure. Here we consider rank-one updates that are linearly invariant and can be efficiently computed by Automatic Differentiation [5]. With this technique it is feasible to compute matrix-vector and vector-matrix products with derivative matrices cheaply.

In the sequel we will drop the index k to simplify the notation. The updated matrices A_{k+1}, B_{k+1} will be denoted by A^+, B^+.

1.1. Updating the Hessian. The Hessian will be modified as shown in [7] by the symmetric-rank-one update formula (SR1):

$$B^+ = B + \frac{(w - Bs)(w - Bs)^T}{(w - Bs)^T s} \equiv B + \epsilon h h^T, \tag{1.2}$$

where

$$w = B^+ s \equiv g(x^+, \lambda) - g(x, \lambda) \tag{1.3}$$

and $s \equiv s_k$ as in (1.1).

It can be seen from (1.3) that this update satisfies the direct secant condition. By Automatic Differentiation the vectors g can be evaluated in the adjoint mode without the knowledge of the full Jacobian $\nabla c(x)$.

1.2. Updating the Jacobian. This matrix can be updated similarly to the Hessian by the two-sided-rank-one update formula (TR1) [7]:

$$A^+ = A + \frac{(y - As)(\mu^T - \sigma^T A)}{\mu^T s - \sigma^T A s} \equiv A + \delta r \rho^T. \tag{1.4}$$

It satisfies the direct secant condition

$$y = A^+ s \equiv c(x^+) - c(x)$$

and the adjoint secant condition up to $\mathcal{O}(\|\sigma\| \|s\|^2)$

$$\sigma^T A^+ = \mu^T \equiv \sigma^T \nabla c(x^+) \tag{1.5}$$

where $\sigma \equiv \sigma_k$ and $s \equiv s_k$ are as in (1.1).

Equation (1.5) can be computed in the reverse or adjoint mode of Automatic Differentiation. Unless the constraint function $c(x)$ is affine the two conditions will not be exactly consistent. But the deviation will only be of order $\mathcal{O}(\|\sigma\| \|s\|)$ which is within the scope of quasi Newton methods. More details about these two updates in this optimization context are given in [7].

ETNA
Kent State University
etna@mcs.kent.edu

1.3. Null-space representation. For the solution of KKT systems it is necessary to solve the linear system of equations (1.1). One way to do this consists of decomposing the matrices A and B. Here and throughout the paper we will assume that A has full rank. In contrast to [8] a complete LU-factorization instead of a QR-decomposition of the approximate Jacobian will be performed.

$$PAQ = LU$$

where $U = \left[\ \underbrace{U_1}_{\in\mathbb{R}^{m\times m}} \quad \underbrace{U_2}_{\in\mathbb{R}^{m\times d}}\ \right]$ with $d = n - m$ and U_1 nonsingular. The factors $P \in \mathbb{R}^{m\times m}$ and $Q \in \mathbb{R}^{n\times n}$ are row- and column permutation matrices.

As in [8] the approximate Hessian will be projected to the null-space of A and the range-space of A^T, respectively. The columns of

$$Z = Q\tilde{Z} \quad \text{with} \quad \tilde{Z} = \begin{bmatrix} -U_1^{-1}U_2 \\ I_d \end{bmatrix}$$

form a basis of the null-space of A. As range-space basis we will use the columns of the permuted identity augmented by a block of zeros

$$Y = Q\tilde{Y} = Q\begin{bmatrix} I_m \\ 0 \end{bmatrix} \in \mathbb{R}^{n\times m}. \tag{1.6}$$

Combining these spaces to $Q = \begin{bmatrix} Y & Z \end{bmatrix}$, (1.1) can be transformed by multiplying from left and right by Q, respectively its transposed to

$$\begin{bmatrix} E & C & U_1^T L^T P_z \\ C^T & M & 0 \\ P_z^T L U_1 & 0 & 0 \end{bmatrix} \begin{bmatrix} s_y \\ s_z \\ \sigma \end{bmatrix} = - \begin{bmatrix} Y^T g \\ Z^T g \\ c \end{bmatrix}, \tag{1.7}$$

with the notation

$$E = Y^T BY \in \mathbb{R}^{m\times m}, \quad C = Y^T BZ \in \mathbb{R}^{m\times d}, \quad M = Z^T BZ \in \mathbb{R}^{d\times d}$$

and $s = Y^T s_y + Z^T s_z$.

Initially we assume that M is positive definite. This can be achieved by starting with identity matrices for A and B. Factorizing the matrix M is necessary to solve system (1.7). To do so we will use a transposed Cholesky factorization: $M = RR^T$ where R is an upper triangular matrix. This is only possible if in every step M is positive definite. We will ensure this, e.g., by damping the (SR1) or (TR1) update. Further note that because of their structures it is not necessary to store Y and Z explicitly. Instead we use the implicit representation via the LU-decomposition of A. Consequently our total storage requirement is the same as that for A and B, and far less than that required by the QR-factorization.

1.4. Updating decompositions. Decomposing the KKT systems (1.1) in the previously described way costs $\mathcal{O}(n^3)$ operations in every Newton step which is quite expensive in the case of large problems. The computational effort can be reduced by one order to $\mathcal{O}(n^2)$ if A and B need not to be re-factorized in every step. Because of the low-rank corrections to A and B by the (SR1) and (TR1) update formulas this is possible by updating the factors directly. Two different procedures for updating the LU-factorization will be presented in detail and compared regarding their costs of computation and their stability. Moreover we will introduce an extension of these algorithms which will further improve the computation time. Numerical examples confirm that this approach is an excellent alternative to algorithms based on orthogonal transformations.

2. LU-updating. The LU-factorization $PAQ = LU$ of a dense matrix $A \in \mathbb{R}^{m\times n}$ has an algebraic complexity of $\mathcal{O}(nm^2)$ operations in general. In particular in our application where we have to solve a sequence of KKT systems the associated submatrices underlie a sequence of low-rank modifications. This requires the factorization of A in every step. Thus it is better to factorize A only once at the beginning of the computation. Then the factors P, Q, L and U can be updated directly with an effort of $\mathcal{O}(mn)$ operations. Next, two different algorithms of Bennett [1] and Schwetlick/Kielbasinski [10] respectively Fletcher/Matthews [2] will be illustrated. Also a new method combining these two algorithms is shown. In addition a new approach for the case of column permutations will be presented. Further we will introduce a faster updating procedure by doing an efficient row-wise implementation and regarding the structure of the low rank term. Moreover it will be shown that the algorithm by Bennett is advantageous in the symmetric positive definite case. The basic problem can be described as follows:

Let the rank-one modification be

$$A^+ = A + uv^T, \tag{2.1}$$

where $u \in \mathbb{R}^m$ and $v \in \mathbb{R}^n$ and let the decomposition

$$PAQ = LU \tag{2.2}$$

be given and assume that A^+ also has full rank. We want to compute new updated factors L^+, U^+, P^+, Q^+ such that

$$A^+ = (P^+)^T L^+ U^+ (Q^+)^T = P^T LUQ^T + uv^T. \tag{2.3}$$

2.1. Algorithm I - Schwetlick/Kielbasinski. At first we introduce the updating algorithm by Schwetlick/Kielbasinski [10]. It can be used for updating an LU-factorization by a rank-one term only using row pivoting. In this method L is a lower triangular matrix with unit diagonal. During the procedure the column permutation Q is kept unchanged. In the case of rectangular matrices A this may cause problems. After updating it can occur that the leading part U_1^+ of U^+ does not have full rank. Then it is necessary to permute a column from the rear part to the front in order to restore the regularity of U_1. A new method which determine and permute appropriate columns will be described in addition to the main algorithm. The updating procedure without column pivoting ($Q \equiv I$) works as follows:

From (2.3) one obtains

$$A^+ = P^T L(U + \tilde{u}v^T) \quad \text{with} \quad P^T L\tilde{u} = u.$$

Using a sequence of elementary transformations, the vector $\tilde{u}$ will be reduced to a multiple of the first vector of unity. This can be done by eliminating the components of $\tilde{u}$ step by step, starting from the last one and going upwards. This procedure requires pivoting since the entries of $\tilde{u}$ may be small. The elimination process looks like

$$\tilde{A} = \underbrace{(P^T P_u^T)}_{\tilde{P}^T}\underbrace{(P_u L T_u^{-1})}_{\tilde{L}}(\underbrace{T_u U}_{\tilde{U}} + T_u\tilde{u}v^T)$$

where

$$T_u = T_{m-1}T_{m-2}\cdots T_1, \quad P_u = P_{m-1}P_{m-2}\cdots P_1.$$

ETNA
Kent State University
etna@mcs.kent.edu

In the simplest case we can find a lower triangular matrix T_i such that

$$A^{(i)} = P^T(LT_i^{-1})(T_iU + T_i\tilde{u}v^T)$$

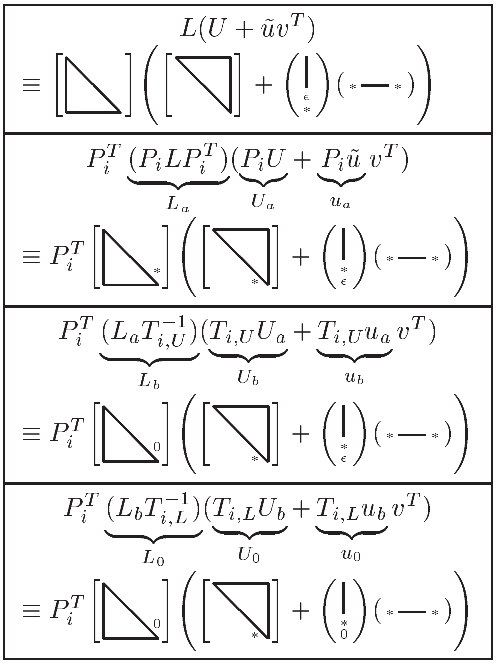
and T_i eliminates $\tilde{u}_{i+1}$. If pivoting is required then $T_i = T_{i,L}T_{i,U}P_i$ is used. These matrices are defined as follows.

Because of the small element ϵ pivoting is required in this example to eliminate the last element of $\tilde{u}$ in a stable way. So we have to chose $T_i = T_{i,L}T_{i,U}P_i$ such that,

a) P_i is chosen to interchange the last two elements $\tilde{u}_i$ and $\tilde{u}_{i+1}$ of $\tilde{u}$,

b) $T_{i,U}$ is the identity matrix plus a single upper diagonal entry which eliminates the addidional element of L_a,

c) $T_{i,L}$ is the identity matrix plus a single sub diagonal entry that finally eliminates the $i+1$-th component of u_b.



Repeating the elimination process finally leads to

$$\tilde{A} = \underbrace{(P^TP_u^T)}_{\tilde{P}^T}\underbrace{(P_uLT_u^{-1})}_{\tilde{L}}(\underbrace{T_uU}_{\tilde{U}} + \underbrace{T_u\tilde{u}}_{\hat{u}}\,v^T)$$

where

$$T_u = T_{m-1}T_{m-2}\cdots T_1, \quad P_u = P_{m-1}P_{m-2}\cdots P_1,$$

and $\tilde{U}$ is upper Hessenberg, $\tilde{P}$ is a permutation matrix and $\tilde{L}$ is lower unit triangular. Since $\hat{u}$ is a multiple of the first unit vector the rank-one term $\hat{u}v^T$ can be added to $\tilde{U}$ without destroying the Hessenberg form. Finally, to eliminate the extra lower sub-diagonal in $\tilde{U}$ a second sequence of transformations has to be done according to the same principle:

$$A^+ = \underbrace{(\tilde{P}^TP_d^T)}_{P^+}\underbrace{(P_d\tilde{L}T_d^{-1})}_{L^+}\underbrace{(T_d\tilde{U})}_{U^+}$$

where

$$T_d = T_1'T_2'\cdots T_{m-1}', \quad P_d' = P_1'P_2'\cdots P_{m-1}'. \tag{2.4}$$

The number of operations amounts to knm for updating an $(m \times n)$ -matrix by this algorithm. Here is $5 \le k \le 9$ where the best case arises if no permutations are required

ETNA
Kent State University
etna@mcs.kent.edu

and the worst bound is attained in the opposite case, i.e., if one has to permute every row.

To achieve numerical stability pivoting is done in [10] if

$$|u_i| < |l_{i+1,i}u_i + u_{i+1}|. \tag{2.5}$$

This condition ensures that $T_{i,L}$ and $T_{i,U}$ always eliminate a small element by a larger one in modulus. Further this guarantees the property that all entries in the first lower sub-diagonal of L remain smaller than one in modulus. Hence all other elements in this matrix can only grow by a factor of three. Certainly, this strategy causes a large number of permutations whereas the runtime of the algorithm is growing. For this reason it is advisable to include a threshold τ in (2.5) for reducing pivoting such that

$$|u_i| < \tau \cdot |l_{i+1,i}u_i + u_{i+1}|$$

where $0 < \tau \leq 1$. As a compromise between numerical stability and algebraic efficiency we suggest to chose $\tau = 0.1$.

REMARK 2.1. In the case that no pivoting is allowed, i.e., $\tau = 0$, it is possible to compute the elements of L^+ row by row. This is a significant point in modern computations where memory accesses needs more and more time compared to floating point operations which become faster. This row-wise computation can be done by storing the transformations T_u and T_d, i.e., storing m elements each. Then the update of L can be done row by row after the corresponding U was modified by T_u and T_d respectively. This modification significantly speeds up the algorithm if $\tau = 0$.

2.2. Column permutations. In the case of updating a rectangular LU-factorization, column permutations in U are required to keep U_1 nonsingular. They are necessary if an element on the main diagonal in U_1 becomes very small or equals zero. In our application we expect that at most one column permutation is necessary in each updating step due to the modification of A is of rank-1. This means only one column $u_i \in U_1$ has to be be interchanged with a suitable column $u_j \in U_2$ per step. This permutation corresponds to

$$U^+ = UP_{ij}$$

where $1 \leq i \leq m < j \leq n$ and P_{ij} interchanges the columns i and j.

Assume that after a rank-one modification the matrix U has the following form:

$$U^{(1)} = \begin{array}{c} \begin{array}{cccccccc} & u_i & & & & & & \end{array} \\ \left[\begin{array}{cccc|cc} \times & \times & \times & \times & \times & \times \\ 0 & \epsilon & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \end{array}\right] \end{array} \equiv \left[U_1 \quad | \quad U_2\right]$$

where $0 \leq |\epsilon| \ll \max\limits_{1\leq i\leq m} (|u_{ii}|)$. Then column u_i can be seen as an approximated linear combination of the columns in front of it. In this case U_1 is almost rank deficient. To restore nonsingularity, u_i is moved to the rear part U_2. The first task is to determine a suitable column $u_j \in U_2$ which can be interchanged with u_i to restore the regularity of the leading part of $U^{(1)}$. Therefore at first the element ϵ must be transformed to the last diagonal position u_{mm}. So we have to remove column u_i and re-insert it in the m -th position. This corresponds to a column permutation in U_1:

$$U^{(2)} = U^{(1)}P^{(1)} = \begin{array}{c} \begin{array}{cccccc} & & & u_i & & \end{array} \\ \left[\begin{array}{cccc|cc} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \epsilon & \times & \times \\ 0 & \times & \times & 0 & \times & \times \\ 0 & 0 & \times & 0 & \times & \times \end{array}\right] \end{array} \tag{2.6}$$

Now $U^{(2)}$ has become upper Hessenberg form and the new lower sub-diagonal entries have to be eliminated with the elementary transformations introduced in Section 2.1.

$$U^{(3)} = TU^{(2)} = \begin{array}{c} \begin{array}{cccccc} & & & \tilde{u}_i & & u_j \end{array} \\ \left[\begin{array}{cccc|cc} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \tilde{\epsilon} & \times & u_{mj} \end{array}\right] \end{array} \tag{2.7}$$

Now the small element ϵ has been moved down in U and we can compare it with the other elements in the last row of U_2. The column u_j of U_2 associated with the largest entry u_{mj} in modulus will be interchanged with $\tilde{u}_i$.

$$U^{+} = U^{(3)}P^{(2)} = \begin{array}{c} \begin{array}{cccccc} & & & u_j & & \tilde{u}_i \end{array} \\ \left[\begin{array}{cccc|cc} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & u_{mj} & \times & \tilde{\epsilon} \end{array}\right] \end{array}$$

Now the matrix U_1 has been transformed to the desired nonsingular form.

The steps (2.6) and (2.7) are only necessary to determine which column of U_2 has to be permuted. The permutation itself can be done in U as the following rank-one update which directly interchange u_i and u_j

$$U^{+} = U + \left(u_i - u_j\right)\left(e_j - e_i\right)^T. \tag{2.8}$$

So it is more advantageous to perform (2.6) and (2.7) on a temporary vector instead explicitly in U. Using the transformations described in (2.4) no further small entries will arise in the main diagonal of U_1 due to the elimination process (2.7).

2.3. Algorithm II - Bennett. The algorithm by Bennett is used to update a triangular factorization by directly changing the factors L and U step by step. All matrices except the permutations are as in (2.1)-(2.3). To our knowledge this procedure can not be combined with pivoting. Of course, this can cause numerical instabilities but offers some great improvements in runtime. Bennetts approach is quite different from Algorithm I. Here the update is done recursively based on:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = L_1 \begin{bmatrix} 1 & 0 \\ 0 & \tilde{A} \end{bmatrix} U_1 = \begin{bmatrix} 1 & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{A} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & I \end{bmatrix},$$

where $\tilde{A} = A_{22} - L_{21}U_{12}$ represents the Schur complement. L_1 is the identity matrix where only the elements on the first column are replaced by corresponding elements

ETNA
Kent State University
etna@mcs.kent.edu

of the first column of L, U_1 is defined analogously. Using this notation we can update the matrix factors row by row and column by column beginning at the top. Given

$$A^+ = A + \gamma u v^T = L_1 \begin{bmatrix} 1 & 0 \\ 0 & \tilde{A} \end{bmatrix} U_1 + \gamma \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}^T$$

we obtain

$$A^+ = \underbrace{\begin{bmatrix} 1 & 0 \\ L_{21}^+ & I \end{bmatrix}}_{L_1^+} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{A}^+ \end{bmatrix} \underbrace{\begin{bmatrix} U_{11}^+ & U_{12}^+ \\ 0 & I \end{bmatrix}}_{U_1^+}$$

where

$$U_{11}^+ = U_{11} + u_1 v_1 \quad , L_{21}^+ = \frac{L_{21} U_{11} + u_2 v_1}{U_{11}^+} \quad , U_{12}^+ = U_{12} + u_1 v_2^T.$$

In addition the new Schur complement $\tilde{A}^+$ can be represented as a remaining rank-one modification of the former Schur complement in the form

$$\tilde{A}^+ = \tilde{A} + \tilde{\gamma} \tilde{u} \tilde{v}^T \tag{2.9}$$

where

$$\tilde{\gamma} = \frac{\gamma}{U_{11}^+}, \quad \tilde{u} = u_2 - L_{21} u_1, \quad \tilde{v}^T = U_{11} v_2^T - v_1 U_{12}.$$

After calculating these terms the procedure can be restarted with the new reduced rank-one update (2.9) for $\tilde{A}$. The new scalar factor γ can be included in one of the vectors $\tilde{u}$, $\tilde{v}^T$. That way the complete updating process can be done step by step obtaining the new factors L^+ and U^+.

The number of operations amounts to $4nm$ for updating an $(m \times n)$-matrix by this algorithm, cp. Fig. 2.1. This is always better than Algorithm I even if there are no permutations necessary. Furthermore, this method can easily be extended for higher rank modifications. In addition, it is applicable to the symmetric positive definite case. Moreover, Bennett's algorithm can be implemented in a way which allows row-wise computation of the new elements of L and U. Similar to Algorithm I where the row-wise memory access can only be done if no pivoting is required it significantly reduces the computational time for updating the matrix L. Therefore the modification of L has to be delayed. This corresponds to the 'ikj-variant' of the Gaussian elimination which is used, e.g., for incomplete LU-factorizations [11]. Fig. 2.1 displays the new row-wise Algorithm in contrast to the standard version of Bennett. In Fig. 2.2 and Fig. 2.3 the differences in matrix access between these two algorithms is shown. The hatching shows which matrix areas have been changed until step i during the updating procedure. Nevertheless the main problem in this algorithm is that there are no known possibilities to combine pivoting with the low rank update. This can cause numerical stability problems during the updating procedure.

REMARK 2.2. In the case of updating the LDL^T factorization of a symmetric positive definite matrix, this method corresponds to the algorithm introduced in [3]. It turns out that this updating technique is an excellent alterative to methods using plane rotations.

ETNA
Kent State University
etna@mcs.kent.edu

Standard recursive LU-updating

```
for i = 1 to m do
   //diagonal update
   U_ii = U_ii + u_i * v_i
   v_i = v_i / U_ii
   for j = i + 1 to m do
      //L update
      u_j = u_j - u_i * L_ji
      L_ji = L_ji + v_i * u_j
   end for
   for j = i + 1 to n do
      //U update
      U_ij = U_ij + u_i * v_j
      v_j = v_j - v_i * U_ij
   end for
end for
```

Row-wise recursive LU-updating

```
for i = 1 to m do
   for j = 1 to i - 1 do
      //delayed L update
      u_i = u_i - u_j * L_ij
      L_ij = L_ij + v_j * u_i
   end for
   //diagonal update
   U_ii = U_ii + u_i * v_i
   v_i = v_i / U_ii
   for j = i + 1 to n do
      //U update
      U_ij = U_ij + u_i * v_j
      v_j = v_j - v_i * U_ij
   end for
end for
```

FIG. 2.1. *Standard and Row-wise Algorithms for the Rank-One Modification of the LU-Factorization*

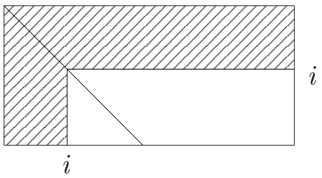


FIG. 2.2. *Standard Bennett Algorithm*

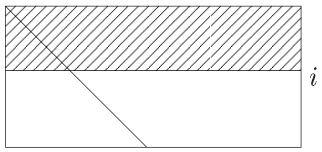


FIG. 2.3. *Row-wise Bennett Algorithm*

2.4. Algorithm III - extension. Here a new combination of the previous described algorithms will be presented. Thereby the good features from both are combined. Furthermore the method will be improved for update vectors beginning with a leading part consisting of zero elements. This is useful in the KKT application introduced in this paper, where linear constraints causes such zero entries.

This algorithm will be done sequentially by the following three stages:

- exploiting leading zeros in u and v to the update
- starting with Algorithm II as long as it is stable
- switching to Algorithm I in the case that pivoting is required

In each step the whole updating problem will be reduced as displayed in Fig. 2.4.

Step 1: If one of the update vectors has a leading block of zero entries, only a submatrix of A has to be modified. We will illustrate the influence on the factors in the case $u^T = \begin{pmatrix} 0 & u_2^T \end{pmatrix}$. Here only the lower part of A will be recomputed:

$$A^+ = \begin{bmatrix} A_1^+ \\ A_2^+ \end{bmatrix} = \begin{bmatrix} A_1 + 0 \cdot v^T \\ A_2 + u_2 v^T \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} + u_2 v_1^T & L_{21}U_{12} + L_{22}U_{22} + u_2 v_2^T \end{bmatrix}.$$

That means that the matrix parts L_{11}, U_{11} and U_{12} remain unchanged. The other

ETNA
Kent State University
etna@mcs.kent.edu

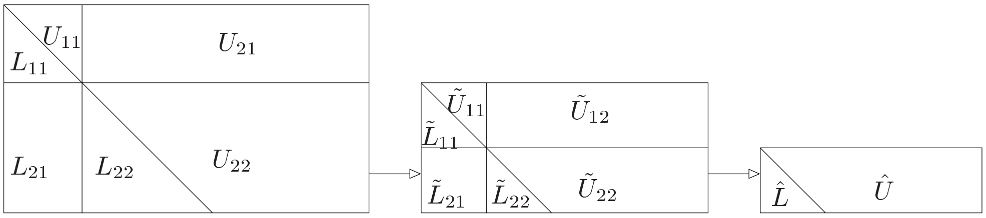


FIG. 2.4. *Algorithm III*

parts can be computed as follows. Given

$$L_{21}^{+}U_{12}^{+} = L_{21}U_{12} + \tilde{u}_2 v_1^T$$

we obtain the formula to calculate L_{21}^{+}

$$(L_{21}^{+} - L_{21})U_{12} = \tilde{u}_p v_1^T \iff L_{21}^{+} = L_{21} + \tilde{u}_p v_1^T U_{11}^{-1}.$$

Now the remaining new sub-matrices L_{22}^{+} and U_{22}^{+} can be computed in form of a new reduced dense LU-updating problem. This is represented by

$$\begin{aligned} L_{21}^{+}U_{12} + L_{22}^{+}U_{22}^{+} &= L_{21}U_{12} + L_{22}U_{22} + \tilde{u}_2 v_2^T \iff \\ (L_{21}^{+} - L_{21})U_{12} + L_{22}^{+}U_{22}^{+} &= L_{22}U_{22} + \tilde{u}_2 v_2^T \iff \\ L_{22}^{+}U_{22}^{+} &= L_{22}U_{22} + \tilde{u}_2 \underbrace{(v_2^T - v_1^T U_{11}^{-1} U_{12})}_{\tilde{v}^T}. \end{aligned} \tag{2.10}$$

From (2.10) we conclude that it is sufficient to consider the reduced updating problem for L_{22} and U_{22}.

This computation can be done in an analogous way if the vector v has leading zero entries.

Step 2: Given the remaining submatrices L_{22}, U_{22} and the update vectors $\tilde{u}_2, \tilde{v}^T$, we initially start using Algorithm II where v_2 is replaced by $\tilde{v}$. As we will see in Section 4, this algorithm will be significantly faster than Algorithm I. For reasons of efficiency we will chose the row-wise version. Algorithm II will be used as long as it is stable (see step 3). If we have to stop it for stability, then L_{21}^{+} is not yet computed. Thereby L_{21} will be modified with respect to $\tilde{L}_{11}, \tilde{U}_{11}$. This will be done by the standard version of Algorithm II.

Step 3: In step i we switch to Algorithm I if

$$|U_{ii}^{+}| \leq \kappa \cdot \max(|U_{i2}^{+}|) \quad \text{where} \quad (0 \leq \kappa \leq 1).$$

This condition is used to safeguard the updating process and it avoids tiny pivots. Up from this point the remaining matrix parts $\hat{L}$ and $\hat{U}$ will be updated by Algorithm I. Notice that row and column permutations applied to $\hat{L}$ and $\hat{U}$ will also be performed on $L_{21}^{+}, \tilde{L}_{21}^{+}, U_{12}^{+}, \tilde{U}_{12}^{+}$.

3. KKT updating. At first we have to modify the approximate Jacobian

$$A^{+} = A + \delta r \rho^T, \tag{3.1}$$

where δ, r and ρ are as in Section 1.2. These terms are computed as part of the optimization process using s and σ according to (1.4) after solving (1.1). Here we assume that A and A^+ have full rank. This can be guaranteed, e.g., by damping the update (3.1). The modification (3.1) can be done by one of the algorithms described in Section 2. Whenever A will be modified we have to adjust its corresponding null-space Z. Due to the regularity of the approximate Jacobian, Z is regular in every step.

3.1. Updating the null-space. If the approximate Jacobian is updated by a rank-one term, the modification of the corresponding null-space of A is of rank-one, too. This rank correction can be computed in a way that Z maintains its trapezoid structure. So we obtain

$$Z^+ = Z + \Delta z \rho_z^T, \quad \text{where } \Delta z \in \mathbb{R}^n, \rho_z \in \mathbb{R}^d. \tag{3.2}$$

Modifications only occur in the matrix product $\hat{Z} = -U_1^{-1}U_2$ of Z. Hence only the upper part of Δz denoted by $\Delta\bar{z} \in \mathbb{R}^m$ is non-zero. These vectors $\Delta\bar{z}$ and ρ_z can be calculated using the Sherman-Morrison-formula [4]. The new significant null-space is given by

$$\hat{Z}^+ = -(U_1^+)^{-1}U_2^+ = -\left[U_1 + \tilde{r}\tilde{\rho}_1^T\right]^{-1}\left[U_2 + \tilde{r}\tilde{\rho}_2^T\right]$$

where $\tilde{\rho}^T = \begin{bmatrix}\tilde{\rho}_1^T & \tilde{\rho}_2^T\end{bmatrix} = \rho^T Q$ and $\tilde{r} = \delta L^{-1}P^T r$. Straightforward computation yields

$$\hat{Z}^+ = \hat{Z}\underbrace{-U_1^{-1}\tilde{r}}_{\Delta\bar{z}}\underbrace{\left[(\tilde{\rho}_2^T) - \frac{(\tilde{\rho}_1^T)U_1^{-1}\left[(U_2) + \tilde{r}(\tilde{\rho}_2^T)\right]}{1 + (\tilde{\rho}_1^T)U_1^{-1}\tilde{r}}\right]}_{\rho_z^T},$$

which represents the null-space modification as rank-one correction corresponding to the update of A. Since a column interchange in A can also be read as a rank-one update, the null-space modification of $\hat{Z}$ can be computed analogously. In the particular case of an additional column interchange in A, it is necessary to collect the two separate rank-one updates into a single rank-two update to avoid numerical problems.

With the knowledge of Δz and ρ_z the projected Hessian can be adjusted with respect to the null-space of A. In the following this will be described in detail.

3.2. Updating the projected Hessian. The projected Hessian has to be modified whenever one of the following three cases occurs:

(a) the SR1 update (1.2): $B_+ = B + \epsilon hh^T$
(b) modification of the null-space (3.2): $\tilde{Z}_+ = \tilde{Z} + \Delta z\rho_z^T$
(c) column permutation in A: $Q^+ = P_{ij}Q$

(a) The matrices E and C are updated with respect to the rank-one correction of B. We obtain

$$E^+ = Y^TB^+Y = Y^T(B + \epsilon hh^T)Y = \underbrace{Y^TBY}_{E} + \epsilon Y^Thh^TY$$

and

$$C^+ = Y^TB^+Z = Y^T(B + \epsilon hh^T)Z = \underbrace{Y^TBZ}_{C} + \epsilon Y^Thh^TZ.$$

The update of the projected null-space $M = Z^T B Z^T$ can be represented by

$$Z^T B_+ Z = R_+ R_+^T = Z^T\big(B + \epsilon h h^T\big)Z = RR^T + \epsilon h_z h_z^T,$$

where $h_z = Z^T h$. As long as the rank-one update is constructed to preserve positive definiteness, it can be computed with several algorithms for updating the Cholesky factorization [3, 4]. Here we have to pay attention to choose ϵ such that M remains positive definite. This can be achieved, e.g., by damping the rank-one term or by pre-adjusting the Hessian [8].

(b) The changes in E and C caused by the null-space modification of Z can be done in a very similar way to case (a). In contrast to the first case, M underlies a rank-two modification

$$R_+ R_+^T = Z_+^T B Z_+ = \big(Z^T + \rho_z \Delta z^T\big) B \big(Z + \Delta z \rho_z^T\big) = \underbrace{Z^T B Z}_{RR^T} + \rho_z b_z^T + b_z \rho_z^T + \mu \rho_z \rho_z^T,$$

where $b_z = Z^T B \Delta z$ and $\mu = \Delta z^T B \Delta z$. Once more we have to avoid to lose positive definiteness of M. This can be achieved using the strategies described in [8].

(c) In the case of column permutations applied to A at first we adjust the projected Hessian similarly to case (b) but use a rank-two term. This rank-two correction consists of the modifications in $\hat{Z}$ caused by the rank-one update of A (1.4) and by the column permutation in U (2.8), compare Section 3.1. Furthermore, the two rows in Q which were interchanged, cause additional changes in E, C and M.

As before the row interchange in the null-space $Q^+\tilde{Z}_+ = P_{ij} Q \tilde{Z}_+$ can be read as a rank-one update which leads us to case (b) again. At last the permutation Q occurring in the range-space basis Y (1.6) has to be regarded, too. This requires to remove and re-compute some single rows, resp. single columns in E and C. We will show this for the matrix C.

Starting with $C = \tilde{Y}^T Q^T B Q \tilde{Z}$ we obtain $C^{(b)} = \tilde{Y}^T Q^T B Q^+ \tilde{Z}_+$ after a low rank correction from the right. Our objective is to compute

$$C^+ = \tilde{Y}(Q^T)^+ B Q^+ \tilde{Z}_+ = \tilde{Y} Q^T P_{ij}^T B Q^+ \tilde{Z}_+.$$

Suppose that $C^{(b)}$ is given, then C^+ can be obtained from $C^{(b)}$ by replacing the i-th row c_i^T by c_j^T, i.e.,

$$C^{(b)} = \begin{bmatrix} c_1^T \\ \vdots \\ \hline c_i^T \\ \hline \vdots \\ c_m^T \end{bmatrix} \Longrightarrow C^+ = \begin{bmatrix} c_1^T \\ \vdots \\ \hline c_j^T \\ \hline \vdots \\ c_m^T \end{bmatrix},$$

where $1 \le i \le m < j \le n$ and c_j^T can be computed as $c_j^T = e_j^T Q^T B Z$. To do so, we need the single row $[e_j^T Q^T B]$ of B. Since B is not explicitly stored we have to use its representation

$$\begin{bmatrix} Y^T \\ Z^T \end{bmatrix} B \begin{bmatrix} Y & Z \end{bmatrix} = \begin{bmatrix} E & C \\ C^T & RR^T \end{bmatrix} \Longrightarrow B = Q \begin{bmatrix} I_m & 0 \\ -\hat{Z}^T & I_d \end{bmatrix} \begin{bmatrix} E & C \\ C^T & RR^T \end{bmatrix} \begin{bmatrix} I_m & -\hat{Z} \\ 0 & I_d \end{bmatrix} Q^T.$$

ETNA
Kent State University
etna@mcs.kent.edu

4. Numerical results. In this section we will compare the algorithms described previously regarding their runtime. The tests were done on an Athlon-XP 2100+ machine with 256kB CPU cache and 512 MB main memory. We implemented the algorithms in C using the operating system Linux and the gcc compiler with the option '-O3'. At first we will compare the computation time of several low-rank update algorithms for rectangular matrices. After that we will solve two KKT problems arising from constrained optimization by the quasi Newton approach of Section 1.

4.1. Rectangular LU-updating. The updating algorithms Alg. I, Alg. II and the QR-updating algorithm of [10] will be compared in runtime. In our computation we started with the identity and applied our algorithms to 50 randomly generated rank-one modifications. The computation times in Table 4.1 and Fig. 4.1 are given in seconds. Alg. I represents the updating method of Schwetlick. It is used in four

TABLE 4.1
Comparing LU-Updating

dimension $\times 10^3$	Alg. I, $\tau = 0$	r-w	Alg. I, $\tau = 1$	#piv	Alg. I, $\tau = 0.1$	#piv	Alg. II	r-w	QR
3×3	49.1	28.2	76.5	274459	51.5	16799	28.7	13.6	90.2
1.5×6	30.4	23.8	45.7	133754	31.1	8230	19.3	14.0	49.8
0.75×12	25.0	22.8	37.2	63880	25.9	3951	16.4	14.0	44.3
6×6	434.2	112.6	635.0	563154	448.0	34573	269.7	55.5	695.1
3×12	177.1	95.6	268.0	275213	182.4	17370	118.1	56.5	271.3
1.5×24	116.4	99.4	165.4	132816	120.3	8235	75.7	58.9	205.6

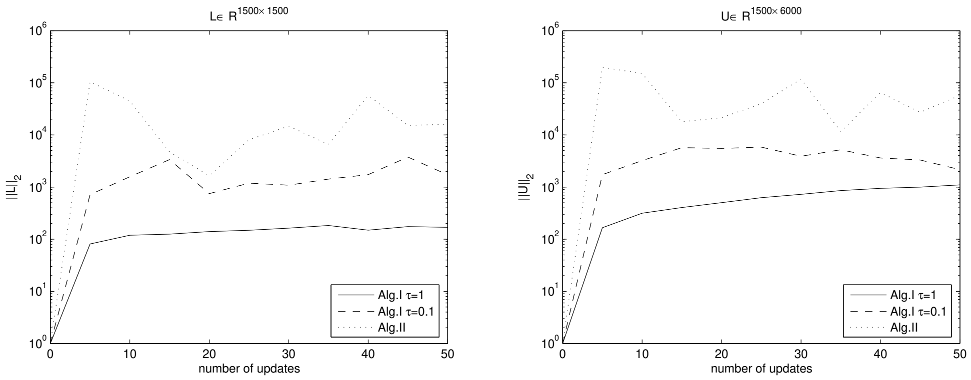


FIG. 4.1. *Comparing LU-Updating from Table 4.1*

different variations. The first two cases use $\tau = 0$ which means that no row pivoting will be done. These two variants differ in the way L is updated. The third version ($\tau = 1$) represents the original algorithm described in [10]. Version 4 of Schwetlick's method uses the relaxed parameter $\tau = 0.1$ which typically reduces the number of row interchanges. For these last two algorithms the total number of permutations is given in the table. Alg. II shows the method of Bennett which is applied in its original form [1] and in the new row-wise implementation, Fig. 2.1. The performance of the QR-updating algorithm is shown in the column 'QR'. In the two diagrams the 2-norms of L and U depending on the number of executed updates are shown for Alg. I ($\tau = 1$), Alg. I ($\tau = 0.1$) and Alg. II.

As we can see, the row-wise version of Alg. II is always the fastest. Especially in the cases of large matrix dimensions it is significantly faster than all other methods. Even the row-wise implementation of Alg. I with $\tau = 0$ which only needs 25% more operations takes at least 40% more time than the row-wise Alg. II. The reason for this is that we have to modify, and this means to access, every element of L and U twice in Alg. I. This takes much more time than recompute every matrix element only once as it is done in Alg. II.

Of course Alg. II offers no possibilities to prevent numerical instabilities by pivoting. As we can see the differences in the norms of L and U are significant depending on which method is used. Among all algorithms that address stability it has turned out that Alg. I with $\tau = 0.1$ performs best. It is approx. 30% faster than in the case where $\tau = 1$ and it is approx. 40% faster than the QR-update.

For these experiments based on random updates we did not use column pivoting since we observed that column interchanges were hardly necessary for this class of problems. For this reason Alg. III which combines Alg. I and Alg. II will be discussed in the next example. Therein we will update a rectangular matrix 50 times by structured rank-one modifications. That means we want to use a sequence of rank-one corrections in the way as we expect in the optimization procedure. Starting with

TABLE 4.2
Comparing Structured Updating

dimension	QR-based	LU-based		
$m \times n$		Alg. I	Alg. II	Alg. III
6000 × 6000	695.10	639.91	55.52	198.23
3000 × 6000	179.15	169.60	27.89	65.27
3000 × 3000	90.20	75.75	9.65	30.82
1500 × 3000	28.51	22.80	6.83	13.45

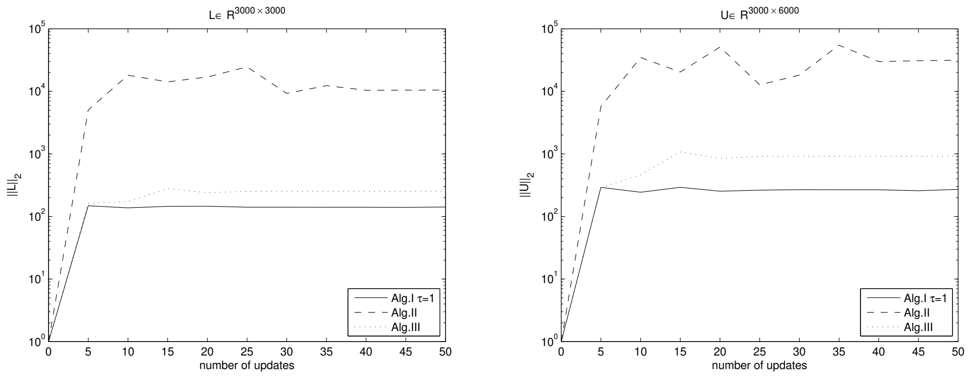


FIG. 4.2. *Comparing Structured Updating from Table 4.2*

vectors which have a large norm leads to much pivoting at the beginning. Step by step we reduce this vector norm so that no permutations will be required finally. We will compare Alg. III with $\kappa = 0.1$ to the QR-based method, to Alg. I using full pivoting and to the fast Bennett method. In Table 4.2 and Fig. 4.2 the runtimes for the total updating processes are presented as well as the norms of L and U. We can see that all LU-based algorithms, Alg. I, Alg. II (row-wise) and Alg. III are faster than

ETNA
Kent State University
etna@mcs.kent.edu

the QR-version. It turns out that only Alg. III can take an advantage of the special structure of the updating sequence. Using the advantages of Alg. I and Alg. II this method is quite fast and offers good stability properties of the updated matrices L and U.

4.2. KKT solving. In this section we will compare methods for solving whole KKT problems. On one hand QR-based algorithms for factorizing and updating A_k together with Givens-techniques for modifying B_k are considered. On the other hand the LU-methods Alg. I, Alg. II and Alg. III applied to A_k are used in combination with the algorithm of [3] for B_k. We use an optimization environment which is provided by Andrea Walther[1]. This code uses a globalizing approach based on line search. For computing the required derivatives the AD-tool ADOL-C [6] is used.

In the first example we will solve the following optimization problem [9]:

Minimize f

$$f(x) = \sum_{i=1}^{n-1} \left(x_i + x_{i+1}\right)^2 \tag{4.1}$$

subject to

$$c_i \equiv x_i + 2x_{i+1} + 3x_{i+2} - 1 = 0 \quad (1 \leq i \leq n-2). \tag{4.2}$$

We initialize x as a random vector where $(-0.5 \leq x_i \leq 0.5)$. The derivative matrices

TABLE 4.3
KKT Example 1

dimension	LU-based			QR-based
n	Alg. I, $\tau = 0.1$	Alg. II	Alg. III	
200	3.75	3.47	3.73	5.23
300	12.12	10.30	12.36	20.64
400	31.45	27.21	32.46	52.61
500	49.83	40.46	51.79	98.71

in the first step are chosen as $A_0 = \begin{bmatrix} I & 0 \end{bmatrix}$ and $B_0 = I$. In Table 4.3 the runtime which was needed to solve (4.1), (4.2) using Alg. I with $\tau = 0.1$, Alg. II row-wise, Alg. III with $\tau = 1, \kappa = 0.1$ and QR-based updating algorithms are shown. Approximately 50% of this time is used for 'non-updating' computations like line search or Algorithmic Differentiation which is nearly identically for all methods. Nevertheless we can see that all LU-based methods are significantly faster than the QR-based version. Because of $n = m + 2$ which means that A is nearly quadratic pivoting is hardly necessary and so all LU-based algorithms lead to the correct result.

In a second example we will solve the following optimization problem [9]:

Minimize f

$$f(x) = \sum_{i=1}^{m} x_{3i-2} + x_{3i-1} + x_{3i-2}x_{3i} + \frac{5}{2}x_{3i} \tag{4.3}$$

[1]Institute for Scientific Computing, D-01062 Dresden, Germany

ETNA
Kent State University
etna@mcs.kent.edu

subject to

$$c_i \equiv \frac{1}{3}x_{3i-2} - x_{3i} = 0 \quad (1 \le i \le m). \tag{4.4}$$

We initialize x as a vector where $x_i = 1$ for $(1 \le i \le 3m = n)$. The derivative matrices in the first step are chosen as $A_0 = \begin{bmatrix} I & 0 \end{bmatrix}$ and $B_0 = I$. Again, in Table 4.4 the runtime which was needed to solve (4.3), (4.4) using Alg. I with $\tau = 0.1$, Alg. II row-wise, Alg. III with $\tau = 1, \kappa = 0.1$ and QR-based updating algorithm is shown.

TABLE 4.4
KKT Example 2

dimension	LU-based			QR-based
n	Alg. I,$\tau = 0.1$	Alg. II	Alg. III	
200	3.14	3.06	3.13	3.93
450	8.92	8.61	9.66	13.51
600	–	–	21.73	32.72

Once again the LU-based methods are faster than the QR-version. Because of the rectangular structure of A performing column pivoting is necessary during the optimization process for some dimensions n. This is the reason why Alg. I and Alg. II break down for $n = 600$. This example shows that it is recommended to use the new hybrid method which combines the fastness of Alg. II with the stability properties of Alg. I.

5. Conclusions. We have introduced several new efficient algorithms for updating LU-factorizations after low-rank modifications. The algorithm of Kielbasinski/Schwetlick was extended for rectangular matrices using additional column pivoting without loosing its quadratic complexity. This is a central point for applying LU-decompositions to non-square systems. Moreover we improved the algorithm of Bennett regarding to efficient matrix access in memory. In addition a new method for special structured updates was developed.

These new algorithms can significantly improve different applications. This is shown in the case of solving nonlinear constraint optimization problems. Numerical results prove that our method is much faster than, e.g., using a QR-based version. So far we have not discussed the case when B_k turns to be indefinite. Currently the updates are sufficiently damped. The generalization to the indefinite case will be discussed in an upcoming paper.

Our promising numerical results indicate that our low-rank-modification algorithms can also be used efficiently in a wide field of applications.

REFERENCES

[1] J. BENNETT, *Triangular factors of modified matrices*, Numer. Math., 7 (1965), pp. 217–221.
[2] R. FLETCHER AND S. MATTHEWS, *Stable modification of explicit LU factors for simplex updates*, Math. Program., 30 (1984), pp. 267–284.
[3] R. FLETCHER AND M. POWELL, *On the modification of LDL^T factorizations*, Math. Comp., 28 (1974), pp. 1067–1087.
[4] G. GOLUB AND C. VAN LOAN, *Matrix Computations*, Second ed., The Johns Hopkins University Press, Baltimore, MD, 1989.
[5] A. GRIEWANK, *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*, Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[6] A. Griewank, D. Juedes, and J. Utke, *Adol-c, a package for the automatic differentiation of algorithms written in c/c++*, ACM Trans. Math. Software 22, (1996), pp. 131–167.
[7] A. Griewank and A. Walther, *On constrained optimization by adjoint based quasi-newton methods*, Optim. Methods Softw., 17 (2002), pp. 869–889.
[8] A. Griewank, A. Walther, and M. Korzec, *Maintaining factorized KKT systems subject to rank-one updates of hessians and jacobians*, Optim. Methods Softw., 22 (2007), pp. 279–295.
[9] W. Hock and K. Schittkowski, *Test Examples for Nonlinear Programming Codes*, Lectures Notes in Econom. and Math. Systems, Springer, New York, 1987.
[10] A. Kielbasinski and H. Schwetlick, *Numerische lineare Algebra*, Verlag Harri Deutsch, 1988.
[11] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing, Boston, 1996.