EFFICIENT SOLUTION OF $A\,x^{(k)} = b^{(k)}$ USING A^{-1}

ADI DITKOWSKI , GADI FIBICH , AND NIR GAVISH

Abstract. In this work we consider the problem of solving $Ax^{(k)} = b^{(k)}$, $k = 1, \ldots, K$ where $b^{(k+1)} = f(x^{(k)})$. We show that when A is a full $n \times n$ matrix and $K \geq cn$, where $c \ll 1$ depends on the specific software and hardware setup, it is faster to solve $Ax^{(k)} = b^{(k)}$ for $k = 1, \ldots, K$ by explicitly evaluating the inverse matrix A^{-1} rather than through the LU decomposition of A. We also show that the forward error is comparable in both methods, regardless of the condition number of A.

AMS subject classifications. 65F05 Direct methods for linear systems and matrix inversion

1. Introduction. One of the first things we learn in a basic numerical linear algebra course is that in order to solve the linear system $Ax = b$, we should not calculate the inverse matrix A^{-1}. For example, we quote from Matlab's user guide:
In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of inverse arises when solving the system of linear equations. One way to solve this is with $x = inv(A) * b$. A better way, from both an execution time and numerical accuracy standpoint, is to use the matrix division operator $x = A\backslash b$. This produces the solution using Gaussian elimination, without forming the inverse.
Similar statements appear in the classical numerical analysis textbooks. For example, Golub and van Loan [7, page 121] say:
... As a final example we show how to avoid the pitfall of explicit inverse computation... when a matrix inverse in encountered in a formula, we must think in terms of solving equations rather than in terms of explicit inverse formation.
Similarly, Conte and de Boor [8, page 166] say:
... given this simple prescription for calculating the inverse of a matrix, we hasten to point out that there is usually no good reason for ever calculating the inverse. ... whenever A^{-1} is needed merely to calculate a vector $A^{-1}b$ (as in solving $Ax = b$) or a matrix product $A^{-1}B$, A^{-1} should never be calculated explicitly.
Higham [9, page 262] also states:
Not only is the inversion approach three times more expensive, but it is much less stable. ... we see that matrix inversion is likely to give a much larger residual than Gaussian elimination with partial pivoting...

Thus, the reasons for preferring Gaussian elimination over explicit calculation of A^{-1} to solve $Ax = b$ are performance, and accuracy and stability.

In this study we consider the case where we want to solve the equations

$$Ax^{(k)} = b^{(k)}, \qquad k = 1, \ldots, K, \tag{1.1}$$

when

1. A is a full $n \times n$ matrix.
2. Equations (1.1) have to be solved sequentially, e.g. when $b^{(k+1)} = f(x^{(k)})$.

The standard, *LU-based approach* involves a preprocessing stage in which the LU decomposition of the matrix A is calculated:

$$[L, U] = lu(A). \tag{1.2}$$

Then, for each right-hand-side the linear system is solved using a forward substitution and a backward substitution:

$$y^{(k)} = L\backslash b^{(k)}; \quad x^{(k)} = U\backslash y^{(k)}, \qquad k = 1, \ldots, K. \tag{1.3}$$

A second, $A^{-1}-$*based approach* for solving problem (1.1) is to calculate the inverse of A in a preprocessing stage:

$$invA = inv(A). \tag{1.4}$$

Then, for each right-hand-side the linear system is solved by a single matrix-vector multiplication:

$$x^{(k)} = invA \cdot b^{(k)}, \qquad k = 1, \ldots, K. \tag{1.5}$$

In this study we show that it is better to solve equations (1.1) using A^{-1} rather than through LU decomposition. In Section 2 we consider the performance issue. We first show that the two methods require the same number of arithmetic operations. However, the actual performance has a lot to do with the way that optimized computer codes, such as Intel's MKL BLAS (Basic Linear Algebra Support) package, handle computer architecture issues, such as caching and memory allocation. Indeed, because the data structure of a square matrix (A^{-1}) is inherently simpler than that of two triangular matrices (L and U), in our numerical tests we observe that it faster to solve these equations using A^{-1} than through the LU decomposition of A. In Section 2.4 we show that there are more efficient implementations of the LU algorithm then the one given in equations (1.2,1.3). Even with these improvements, however, the A^{-1} algorithm is still more efficient.

As noted, the second argument against using A^{-1} is accuracy and stability. This argument is based on analysis which showed that the backward error in the $A^{-1}-$based method is larger than in the LU-based method, and that the ratio between the two increases with the condition number of A. While there has been no such analytical results for the forward error, it was implicitly assumed that it would behave similarly. However, in Section 3, our simulations with randomly-chosen matrices, as well as with three families of ill-conditioned matrices, suggest that this implicit assumption is wrong. Indeed, we observe that the *forward error is of the same magnitude for both the $A^{-1}-$based method and the LU-based method, regardless of the condition number of A.*

In Section 4 we list various applications that are expected to run faster when implemented with the $A^{-1}-$based method (quasi-Newton iterations, inverse power method, and spectral differentiation methods). We conclude by showing that the inverse power method for finding the smallest eigenvalue of a 100×100 matrix can be 11 times faster with the $A^{-1}-$based method than with the LU-based method, and just as accurate.

Our conclusion that using A^{-1} is superior to using LU decomposition applies to problems in which:

1. A is full, since when A is sparse then L and U are sparse but A^{-1} is full.
2. Equations (1.1) have to be solved sequentially, hence only level 2 BLAS optimization (matrix-vector operations) can be used. Indeed, when all the $b^{(k)}$ are known in advance, all the right-hand sides can be simultaneously solved with a block LU algorithm, i.e., replacing (1.3) with

$$Y = L \backslash B; \quad X = U \backslash Y,$$

where

$$B = [b^{(1)} \cdots b^{(K)}], \quad X = [x^{(1)} \cdots x^{(K)}], \quad Y = [y^{(1)} \cdots y^{(K)}].$$

In this case, the block LU algorithm is more efficient than using $X = A^{-1} \cdot B$, since one can exploit level 3 BLAS optimization (matrix-matrix operations).

2. Performance.

2.1. Arithmetic operations count. We begin with the standard count of arithmetic operations:

LEMMA 2.1. *The overall number of arithmetic operations needed to solve the system (1.1) in the LU-based method is*

$$N_{lu} = \underbrace{\frac{2}{3}n^3 + \mathcal{O}(n^2)}_{eq.\ (1.2)} + \underbrace{2Kn^2 + \mathcal{O}(Kn)}_{eq.\ (1.3)},$$

and in the $A^{-1}-$based method

$$N_{A^{-1}} = \underbrace{\frac{8}{3}n^3 + \mathcal{O}(n^2)}_{eq.\ (1.4)} + \underbrace{2Kn^2 + \mathcal{O}(Kn)}_{eq.\ (1.5)}.$$

Proof. For the LU-based method the preprocessing stage involves finding the LU decomposition of A, which requires $\frac{2}{3}n^3 + \mathcal{O}\left(n^2\right)$ arithmetic operations, see [2, page 152]. Then, solving each linear system involves a backward and forward substitution. Each substitution requires $n^2 + \mathcal{O}(n)$ arithmetic operations. Therefore, an overall of $2Kn^2 + \mathcal{O}(Kn)$ arithmetic operations are needed to solve (1.3).

Similarly, for the $A^{-1}-$based method the preprocessing stage involves finding A^{-1}, which requires $\frac{8}{3}n^3 + \mathcal{O}\left(n^2\right)$ arithmetic operations, see [2, page 161]. Then, solving each linear system requires multiplying $b^{(k)}$ by A^{-1}, which requires $2n^2 + \mathcal{O}(n)$ arithmetic operations. Therefore, an overall of $2Kn^2 + \mathcal{O}(Kn)$ arithmetic operations are needed to solve (1.5). $\square$

Observation 2.1 shows that the number of arithmetic operations for the LU-based method is smaller than for the $A^{-1}-$based method for $K \leq n$, and is the same for $K \gg n$. More precisely,

$$\begin{cases} N_{lu} \approx \frac{1}{4} N_{A^{-1}} & K \ll n \\ N_{lu} = \frac{2+6\lambda}{8+6\lambda} N_{A^{-1}} & K = \lambda n, \quad \lambda = \mathcal{O}(1) \\ N_{lu} \approx N_{A^{-1}} & K \gg n. \end{cases} \tag{2.1}$$

2.2. Numerical tests. The standard approach to estimate run-time of algorithms is to count the number of arithmetic operations. Hence, Lemma 2.1 and equation (2.1) seem to suggest that the LU-based method is faster than the $A^{-1}-$based method. We now show that this is not always the case.

To demonstrate this, we solved linear systems of the form (1.1) using the LU-based and $A^{-1}-$based methods. For each method and for each $n = 2^j$, $j = 5, \ldots, 12$, we randomly choose 3 different $n \times n$ matrices, for each matrix we solve the linear system $Ax = b^{(k)}$ 1000 times, sequentially, and calculate the average CPU time per RHS. This measurement of CPU time does not include the preprocessing time of LU decomposition or of finding the inverse.

The tests were conducted on a Pentium 4 with 1Mb cache and 2GB RAM using Matlab 7.0 R14 for Linux. In the first test we used Intel's MKL BLAS (Basic Linear Algebra Support) package. The average CPU time per RHS with the $A^{-1}-$based method is smaller by a factor of 15-35 compared with the LU-based method (see Figure 2.1), for matrix sizes varying from 32×32 to 4096×4096.

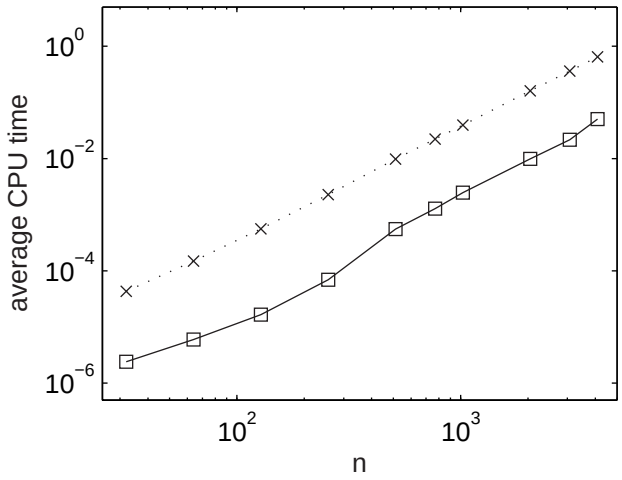

FIG. 2.1. *Average CPU time for solving $Ax = b^{(k)}$ using the backward and forward substitution (eq. (1.3), $\times$) and through multiplication by A^{-1} (eq. (1.5),).*

Next, we repeated the same test, but changed the software package that Matlab uses for the linear algebra subroutines from Intel's MKL BLAS to ATLAS BLAS. In this case, the average CPU time for a forward and a backward substitution did not change, but the average CPU time for multiplication by A^{-1} increased by a factor of $\approx 1.2 - 1.9$, hence the $A^{-1}-$based method was "only" 8-16 times faster than the LU-based method, see Figure 2.2.

Finally, we ran a benchmark test in which we implemented the multiplication by A^{-1} and the backward and forward substitution using a naive straightforward non-vectorized code of a double loop. In that case, the average CPU time is about the same for forward and backward substitution and for multiplication by A^{-1}, see Figure 2.3. As expected, the naive implementation was slower than the MKL BLAS implementation. The matrix-vector multiplication by A^{-1} was 35-50 times faster (using MKL BLAS) than with the naive

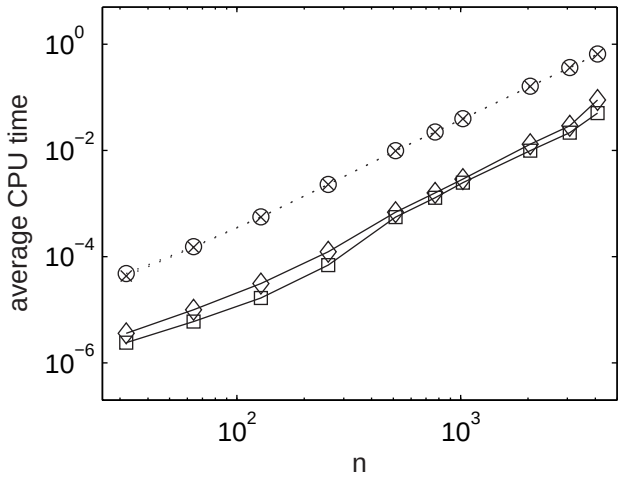

FIG. 2.2. *Same data as in Figure 2.1. Also plotted, is the average CPU time using ATLAS BLAS for solving* $Ax = b^{(k)}$ *using the forward and backward substitution (eq. (1.3),* $\circ$*) and through multiplication by* A^{-1} *(eq. (1.5,* $\diamond$*).*

implementation. On the other side, forward and backward substitution using MKL BLAS was only 1.4-2.6 times faster than forward and backward substitution with the naive implementation.

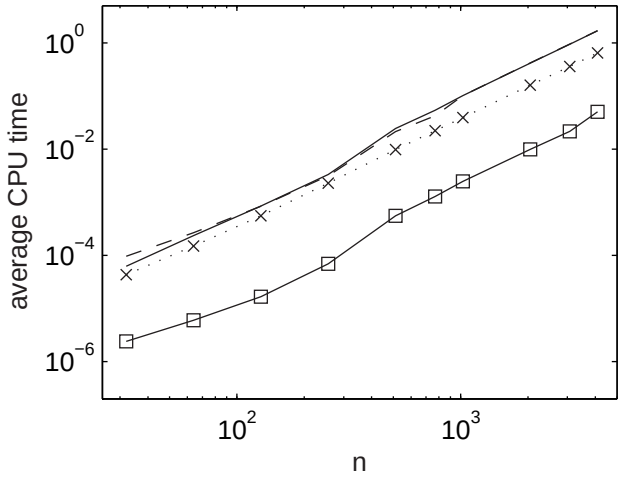

FIG. 2.3. *Same data as in Figure 2.1. In addition average CPU time per RHS using a naive implementation for the LU-based method (solid) and the* $A^{-1}-$*based method (dashes).*

We also conducted the same tests on a different Pentium 4 machine (0.5Mb cache and 0.5GB RAM using Matlab 7.0 R14 for Windows XP). Under this configuration, all qualitative results remained the same, but the performance ratios changed. For example, the $A^{-1}-$based method was "only" 15-20 times faster than the LU-based method using Intel's MKL BLAS, and 8-12 faster under ATLAS BLAS.

2.3. Data analysis. The simulation of Section 2.2 shows that, for sequentially solving the system and neglecting the preprocessing costs, the $A^{-1}-$based method is significantly faster than the LU-based method (per RHS). Since the number of arithmetic operations per RHS is the same for both methods (see Section 2.1), we conclude that the $A^{-1}-$based method is faster since the implementation of matrix multiplication in MKL BLAS or in ATLAS BLAS is faster than the implementation of a forward and backward substitution. Indeed, when we used a naive implementation, there was no performance difference between the two methods. In retrospect, the fact that the $A^{-1}-$based method is faster is not surprising since the memory structure and indexing of a full matrix is inherently simpler than that of a triangular matrix. Therefore, matrix

multiplication can be more easily accelerated through software and hardware optimization. In other words, the performance difference between the two methods is due to the implementation of the numerical linear algebra software package. This conclusion is further supported by the observation that changing the BLAS package leads to a change in the performance ratio from 15-35 to 8-16 , see Figure 2.2.

In the numerical experiments shown so far, we neglected the preprocessing cost and considered only the cost of the solving the linear systems (1.3) and (1.5). This corresponds to the case where $K \gg n$, since then the preprocessing costs are negligible. We now take into account the preprocessing cost and look at the overall cost of solving (1.1) in both methods. For example, setting K=10 and comparing the overall cost (preprocessing+solving for 10 RHS) of the two methods, we obtain that the $A^{-1}-$based method is faster that the LU-based method for matrices of size $n < n_{th}$ and slower for $n > n_{th}$, where the threshold is $n_{th}(K = 10) = 700$, see Figure 2.4. Equivalently, for a given n we can find a threshold $K_{th} = K_{th}(n)$, such the overall cost for solving (1.1) is faster for the $A^{-1}-$method when $K > K_{th}$ and slower when $K < K_{th}$.

There is a linear relation between K_{th} and n. Indeed, the overall cost of the LU-based method is

$$T_{lu}^{preprocessing} + K \cdot T_{lu}^{RHS} = \alpha_{lu}^{preprocessing} n^3 + K \cdot \beta_{lu}^{RHS} n^2 + \mathcal{O}(Kn, n^2), \tag{2.2}$$

where $T_{lu}^{preprocessing}$ is the preprocessing cost of (1.2) and T_{lu}^{RHS} is the cost of the solving the linear systems (1.3) by forward and backward substitution. The overall cost of the $A^{-1}-$based method is

$$T_{inverse}^{preprocessing} + K \cdot T_{inverse}^{RHS} = \alpha_{inverse}^{preprocessing} n^3 + K \cdot \beta_{inverse}^{RHS} n^2 + \mathcal{O}(Kn, n^2), \tag{2.3}$$

where $T_{inverse}^{preprocessing}$ is the preprocessing cost (1.4) and $T_{inverse}^{RHS}$ is the cost of multiplication by A^{-1}, see (1.5). Hence,

$$\begin{aligned} K_{th} &= \frac{T_{inverse}^{preprocessing} - T_{lu}^{preprocessing}}{T_{lu}^{RHS} - T_{inverse}^{RHS}} \\ &= \frac{(\alpha_{inverse}^{preprocessing} - \alpha_{lu}^{preprocessing}) n^3 + \mathcal{O}(n^2)}{(\beta_{lu}^{RHS} - \beta_{inverse}^{RHS}) n^2 + \mathcal{O}(n)} \sim \frac{\alpha_{inverse}^{preprocessing} - \alpha_{lu}^{preprocessing}}{\beta_{lu}^{RHS} - \beta_{inverse}^{RHS}} n + \mathcal{O}(1). \end{aligned} \tag{2.4}$$

In Figure 2.5 we plot the value of K_{th} as a function of n. The results show that $K_{th} \sim 0.0012n + 1.25$, in agreement with (2.4).

As noted, Observation 2.1 has been traditionally interpreted to imply that the LU-based method is faster than the $A^{-1}-$based method for $K = \mathcal{O}(n)$ and that the run time of the two methods is the same for $K \gg n$. In contrast, in Figure 2.5 we see that the $A^{-1}-$based method is faster for $K > cn$, where $c \approx 0.012$. The reason for this disagreement is the implicit assumption that the number of arithmetic operations is a good measure for comparison of run-times. This assumption fails, however, to capture the large difference between $\beta_{inverse}^{RHS}$ and β_{lu}^{RHS} that results from the difference in implementation of matrix-vector multiplication and of forward and backward substitutions.

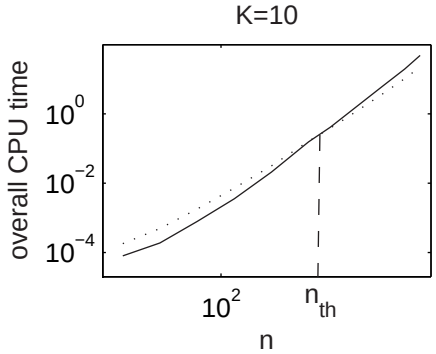

FIG. 2.4. *Overall cpu time of the LU-based method (dots) and the $A^{-1}-$based method (solid) with $K = 10$.*

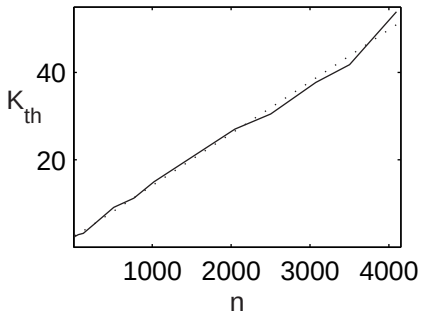

FIG. 2.5. *K_{th} as a function of n_{th} (solid). Dotted line is the fitted linear curve $K_{th} = 0.0124n + 1.25$*

2.4. Improving the LU solver. In our numerical tests so far we used the most straightforward implementation of the LU algorithm for solving (1.1). We first calculated the LU decomposition (1.2). Then, for each right-hand-side we solved the linear system using a forward substitution and a backward substitution, see equation (1.3), which are implemented by a backslash ('\') or equivalently by the 'mldivide' Matlab command.

As stated by the Matlab's help, the command

$$[L, U] = lu(A);$$

returns an upper triangular matrix in U, and a "psychologically lower triangular" matrix (i.e., a potentially permuted lower triangular matrix) in L. The "psychologically lower triangular" form makes it harder to efficiently implement forward substitution. A faster way is, therefore, to enforce L to be a strict lower triangular matrix by using the command

$$[L, U, P] = lu(A), \tag{2.5}$$

where P is the permutation matrix. Then, the solution to $Ax = b$ can be obtained by

$$\begin{aligned} &y = L\backslash(P * b); \\ &x = U\backslash y; \end{aligned} \tag{2.6}$$

The additional calculations in this modification (i.e., the calculation of $P * b$) are $\mathcal{O}(n)$, hence are negligible compared with the time saved by forcing L to be triangular. Indeed, Figure 2.6 shows that using the strict lower triangular matrix is about 2.5 times faster than the standard method.

The backslash command, e.g. $A\backslash b$, involves a preprocessing stage in which the properties of the matrix A are checked in order to pick the most appropriate solver for the problem. Thus, in the case of equation (2.6), the backslash command identifies that L and U are triangular matrices and uses backward- or forward-substitution to obtain the solution. Since it is known that L and U in (2.6) are lower- and upper-triangular matrices, it is possible to bypass the preprocessing stage by using the 'linsolve' command instead of the backslash command. Therefore, we use (2.5) to obtain a strict lower triangular matrix L, but replace (2.6) with

$$\begin{aligned} &optLT.LT = true; \quad \% \quad \text{lower triangular property} \\ &optUT.UT = true; \quad \% \quad \text{upper triangular property} \\ &y = linsolve(L, P * b, optLT); \\ &x = linsolve(U, y, optUT); \end{aligned} \tag{2.7}$$

Figure 2.6 shows that using (2.5) and (2.7) is 4.6 times faster than the standard method, i.e. equations (1.2) and (1.3), and 1.3 times faster than using a strict lower triangular matrix without linsolve, i.e, (2.5) and (2.6). *Even after all these improvements, however, the A^{-1} method is still considerably faster.*

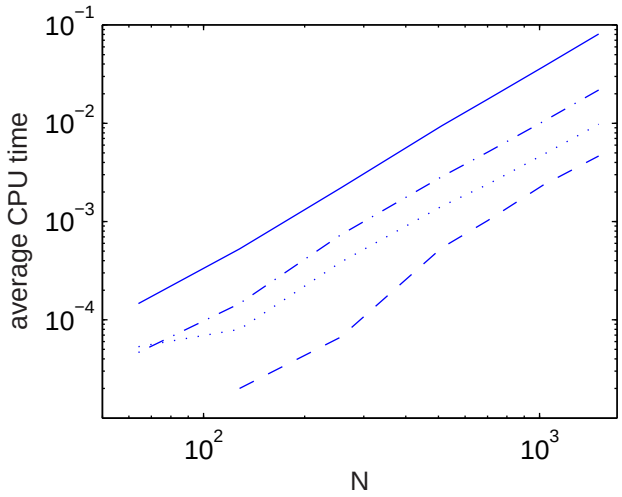

FIG. 2.6. *Average CPU time for solving $Ax = b^{(k)}$ using standard LU method (solid), LU with explicit pivot matrix (dash-dotted), LU and linsolve (dotted) and $A^{-1}-$method (dashed).*

3. Accuracy and stability. The second common argument in the Literature against solving $Ax = b$ with A^{-1} is its numerical accuracy and stability (see citations in the Introduction). As noted, this argument is based on analysis which showed that the backward error in the $A^{-1}-$based method is larger than in the LU-based method, and that the ratio between the two increases with the condition number of A. While there has been no such analytical results for the forward error, it was implicitly assumed that it would behave similarly. Our simulations however suggest that this implicit assumption is wrong. Indeed, we observe that the *forward error is of the same magnitude for both the $A^{-1}-$based method and the LU-based method, regardless of the condition number of A.*

3.1. Forward error. In the $A^{-1}-$based method,

$$\frac{\|\Delta x\|}{\|x\|} \leq cond(A)\frac{\|\Delta b\|}{\|b\|}.$$

In the LU-based method x is determined by solving the linear systems (1.3), hence,

$$\frac{\|\Delta x\|}{\|x\|} \leq cond(L)cond(U)\frac{\|\Delta b\|}{\|b\|}.$$

Since $cond(A) \leq cond(L)cond(U)$, the worst case relative (forward) error $\frac{\|\Delta x\|}{\|x\|}$ is likely to be *larger* for the LU-based method than for the $A^{-1}-$based method. On the other hand, the roundoff error involved in matrix vector multiplication is greater than the roundoff error of backward and forward substitution [9, page 262].

We now show that, in practice, the relative error of the $A^{-1}-$based method is only slightly larger than the relative error of the LU-based method. To do so, we first solved linear systems of the from (1.1) using the LU-based and $A^{-1}-$based methods. For each $n = 100, 200, \cdots, 4000$, we randomly choose 25 different $n \times n$ matrices and a solution vector x. For each matrix A, we first generate the vector $b = Ax$, then solve with each method $A\hat{x} = b$, calculate the relative forward error $\frac{\|\hat{x}-x\|_\infty}{\|x\|_\infty}$ and then average over all 25 matrices. Results in Figure 3.1A show that the forward error of the LU-based is smaller by a factor of 2-4 than for the $A^{-1}-$based method. This 2-4 factor does not change significantly as n changes from 100 to 4000, and $cond(A)$ varies between 10^3 to 10^6. In order to see whether larger differences would appear for ill-conditioned matrices, we perform the same test with Matlab's matrix gallery of involutory ill-conditioned matrices (Figure 3.2A). In this case, the relative forward error for the LU-based method is slightly below the error for the $A^{-1}-$based method, differing by a factor of $1-1.5$, as $cond(A)$ varies between $1-10^{16}$. Repeating this test for tridiagonal ill-conditioned matrices and asymmetric ill-conditioned matrices gave nearly identical results, with factors of 1-2 and 1.4-3.5, respectively.

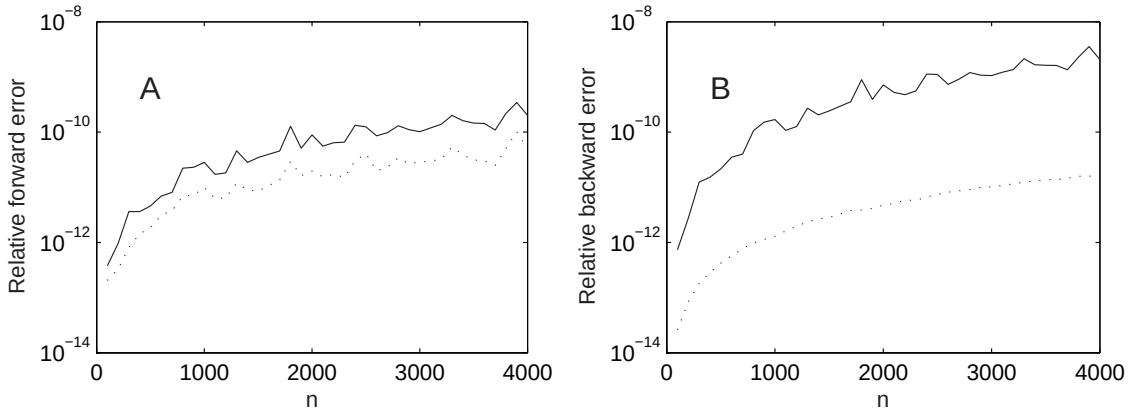

FIG. 3.1. *A: Relative forward error* $\frac{\|\hat{x}-x\|_\infty}{\|x\|_\infty}$ *for* $\hat{x} = A^{-1}b$ *(solid) and* $\hat{x} = A\backslash b$ *(dots). B: Relative backward error for the same data.*

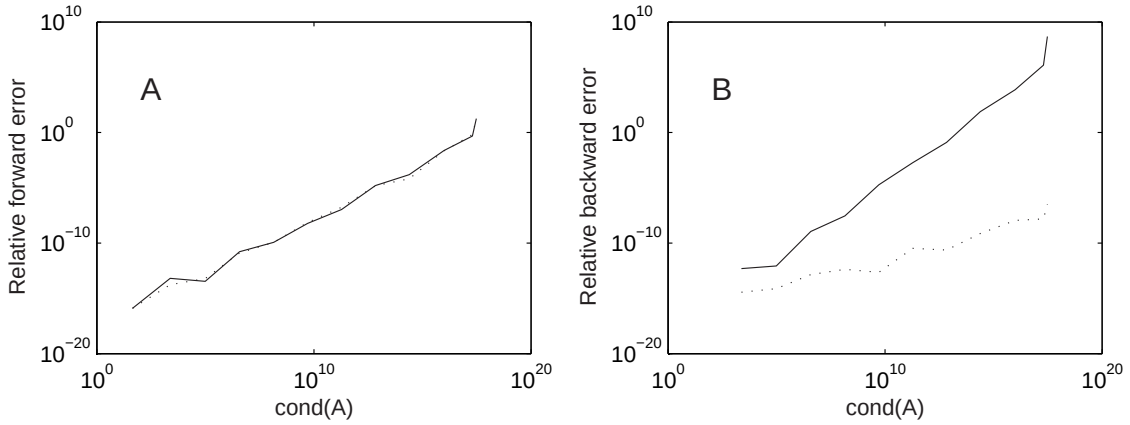

FIG. 3.2. *A: Forward error* $\frac{\|x-\hat{x}\|_\infty}{\|x\|_\infty}$ *for* $\hat{x} = A^{-1}b$ *(solid) and for* $\hat{x} = A\backslash b$ *(dots). The two lines are almost indistinguishable. B: Backward error* $\frac{\|A\hat{x}-b\|_\infty}{\|b\|_\infty}$ *for the same data.*

3.2. Backward error. The backward error $\|A\hat{x}-b\|$ is known to be significantly larger in the $A^{-1}-$based method than in the LU-based method when A is ill-conditioned [9, page 262]. Indeed, Figure 3.1B shows that the backward error of the LU-based method is smaller by a factor of 50-200 than for the $A^{-1}-$based method, for the same matrices used to generate Figure 3.1A. Similarly, Figure 3.2B shows that the backward error of the LU-based method becomes exponentially smaller than for the $A^{-1}-$based method, for the same ill-conditioned involutory matrices used to generate Figure 3.2A.

These results, thus, confirm that the LU-based method is superior to the $A^{-1}-$based method in terms of backward error. However, comparison between Figures 3.1A and 3.1B and between Figures 3.2A and 3.2B shows, that the backward error is a poor prediction to the forward error.

Remark: If the matrix A can be diagonalized, the worst case scenario is that b is the eigenvector of the maximal eigenvalue (in absolute value) and Δb is proportional to the eigenvector of the minimal eigenvalue. However, in most applications arising from PDE, this is not the case. The main 'mass' is concentrated in the lower modes, i.e. in the eigenvector correlated to the lower eigenvalues, while the 'noise' is related to the high frequency modes. Therefore in such applications the worst case scenario is unlikely to happened.

4. Potential Applications. Any application that involves solving many linear systems with a constant *full* matrix A can benefit from adopting the $A^{-1}-$based approach over the LU-based method. We now list various applications which are potentially suitable for the $A^{-1}-$based method:

1. **Spectral differentiation methods for PDEs:** While finite difference methods require solving linear systems with a sparse banded matrix, spectral methods, in many cases, require solving of linear systems with full matrices [3]. For example, when solving a parabolic PDE in an implicit method, each time step involves solving a linear system of the form

$$Dx^{(t+\Delta t)} = F(x^{(t)}),$$

where D is a full matrix which does not vary with time t. In this case, if the number of time steps $K > K_{th}(n)$, it is faster to adopt the $A^{-1}-$based approach, i.e., we invert D at the beginning of the simulation, and then at each step calculate

$$x^{(t+\Delta t)} = D^{-1}F(x^{(t)}).$$

2. **Modified Newton methods:** Newton methods are used to find the roots of nonlinear equations. They involve the iterations

$$x^{(k+1)} = x^{(k)} - [J^{(k)}]^{-1}F\left(x^{(k)}\right), \tag{4.1}$$

where $J^{(k)} = J\left(x^{(k)}\right)$ is the Jacobian of the function F at the point $x^{(k)}$. Hence, $x^{(k+1)}$ is the solution of the linear system

$$J^{(k)}x^{(k+1)} = b^{(k)}, \qquad b^{(k)} = J^{(k)}x^{(k)} - F\left(x^{(k)}\right). \tag{4.2}$$

The calculation of the Jacobian at every iteration is typically very expensive. To reduce the cost of the Jacobian calculation, in the modified Newton method [6], the Jacobian is calculated only once every several (K) iterations.
The common approach is to avoid the explicit calculation of J^{-1}, and find $x^{(k+1)}$ from (4.2). Our analysis shows that it is faster to use the $A^{-1}-$based approach and calculate $x^{(k+1)}$ from (4.1) for "small" matrices of size $n < n_{th}(K)$.
A similar algorithm which may benefit from applying the $A^{-1}-$based method is the Update Skipping BFGS algorithm used in optimization problems [1]. In this case, the iterations of the BFGS method are

$$B_k x^{(k+1)} = B_k x^{(k)} - \lambda J\left(x^{(k)}\right), \tag{4.3}$$

where B_k is an approximation to the Hessian of F at the point $x^{(k)}$ [5]. To reduce the cost of the Hessian calculation, in the Update Skipping BFGS algorithm, B_k is updated only once every K steps. In this case, Our analysis shows that it is faster to use the $A^{-1}-$based approach and calculate $x^{(k+1)}$ from

$$x^{(k+1)} = x^{(k)} - \lambda_k B_k^{-1} J\left(x^{(k)}\right), \tag{4.4}$$

for matrices of size $n < n_{th}(K)$.

3. **Inverse iteration:** The smallest eigenvalue of a full matrix A (and the corresponding eigenvector) can be found from the sequence $\frac{\|v^{(k+1)}\|}{\|v^{(k)}\|}$, where [2],

$$Av^{(k+1)} = v^{(k)}. \tag{4.5}$$

The common approach is to calculate $v^{(k+1)}$ using the LU-based method, i.e.,

$$Ly = v^{(k)}, \quad Uv^{(k+1)} = y. \tag{4.6}$$

However, in this case, calculating $v^{(k+1)}$ using the $A^{-1}-$based approach

$$v^{(k+1)} = A^{-1}v^{(k)} \tag{4.7}$$

is faster if the number of iterations $K > K_{th}(n)$.

Method	Equations	CPU time (seconds)
$[L, U]$	(1.2), (1.3)	19.76
$[L, U, P]$	(2.5), (2.6)	7.6
$[L, U, P]$ + linsolve	(2.5), (2.7)	4.62
A^{-1}	(1.4), (1.5)	1.88

TABLE 4.1
Comparison of CPU run times for the inverse iteration method.

As a final example, we implement the inverse power method (4.5) using both methods as follows. We randomly choose 250 random 100×100 matrices. For each matrix we measure the *overall* cpu-time (preprocessing and iterations) needed to calculate the smallest eigenvalue using the inverse power method implemented with the $A^{-1}-$based method (4.7) and the LU-based method (4.6). We stop the iterations when the relative error is below 10^{-14} (with respect to the smallest eigenvalue found by Matlab's *eig* function).

In our simulations we observed that the $A^{-1}-$based method was about *11 times faster* than the LU-based method (see Table 4.1). The average number of iterations needed for the iterations to converge was 269 in $A^{-1}-$based method and 264 in the LU-based method. This 2% difference is probably due to the larger roundoff errors in the $A^{-1}-$based method. Therefore, in this case, the $A^{-1}-$based method is significantly faster than the LU-based method, and just as accurate. Using the "accelerated" LU algorithms (see Section 2.4) lead to a considerable reduction in the overall CPU time, but was still 2.5 times slower than using A^{-1}.

Acknowledgments. We thank Eli Turkel and Sivan Toledo for useful discussions.

REFERENCES

[1] J.E. Dennis and Jr. Schnabel and B. Robert, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, New York, 1983.
[2] N.L. Trefethen and D. Bua, *Numerical Linear Algebra*, Siam, Philadelphia, 1997.
[3] N.L. Trefethen, *Spectral methods in MATLAB*, Siam, Philadelphia, 2000.
[4] C.T. Kelley, *Solving Nonlinear Equations with Newton's Method*, Siam, Philadelphia, 2003.
[5] K.G. Tamara and D. P. O'Leary and L. Nazareth, BFGS with update skipping and varying memory,SIAM J. Optim.,Vol. 8, 1998.
[6] A. Quarteroni and R. Sacco and F. Saleri, Numerical Mathematics, Springer, New-York, 2000
[7] G. Golub and C. van Loan, Matrix computations, 2rd ed., The Johns Hopkins University Press, London, 1989.
[8] S. D. Conte and C. de Boor, Elementary Numerical Analysis, 3rd ed., McGraw-Hill book company, 1980.
[9] N. J. Higham, Accuracy and Stability of Numerical Algorithms, Siam, Philadelphia, 1996.