MIT Libraries | DSpace@MIT

MIT Open Access Articles

Blendenpik: Supercharging LAPACK's Least-Squares Solver

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Avron, Haim, Petar Maymounkov, and Sivan Toledo. "Blendenpik: Supercharging LAPACK's Least-Squares Solver." SIAM Journal on Scientific Computing 32.3 (2010): 1217. c2010 Society for Industrial and Applied Mathematics

As Published: http://dx.doi.org/10.1137/090767911

Publisher: Society for Industrial and Applied Mathematics

Persistent URL: http://hdl.handle.net/1721.1/60954

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of Use: Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.

Massachusetts Institute of Technology

SIAM J. SCI. COMPUT.
Vol. 32, No. 3, pp. 1217–1236
© 2010 Society for Industrial and Applied Mathematics

BLENDENPIK: SUPERCHARGING LAPACK'S LEAST-SQUARES SOLVER*

HAIM AVRON†, PETAR MAYMOUNKOV‡, AND SIVAN TOLEDO†

Abstract. Several innovative random-sampling and random-mixing techniques for solving problems in linear algebra have been proposed in the last decade, but they have not yet made a significant impact on numerical linear algebra. We show that by using a high-quality implementation of one of these techniques, we obtain a solver that performs extremely well in the traditional yardsticks of numerical linear algebra: it is significantly faster than high-performance implementations of existing state-of-the-art algorithms, and it is numerically backward stable. More specifically, we describe a least-squares solver for dense highly overdetermined systems that achieves residuals similar to those of direct QR factorization-based solvers (LAPACK), outperforms LAPACK by large factors, and scales significantly better than any QR-based solver.

Key words. dense linear least squares, randomized numerical linear algebra, randomized preconditioners

AMS subject classifications. 65F20, 68W20, 65F10

DOI. 10.1137/090767911

1. Introduction.

Randomization is arguably the most exciting and innovative idea to have hit linear algebra in a long time. Several such algorithms have been proposed and explored in the past decade (see, e.g., [23, 10, 9, 22, 17, 12, 21, 8, 5] and the references therein). Some forms of randomization have been used for decades in linear algebra. For example, the starting vectors in Lanczos algorithms are always random. But recent research led to new uses of randomization: random mixing and random sampling, which can be combined to form random projections. These ideas have been explored theoretically and have found use in some specialized applications (e.g., data mining [15, 5]), but they have had little influence so far on mainstream numerical linear algebra.

Our paper answers a simple question, Can these new techniques beat state-of-the-art numerical linear algebra libraries *in practice*?

Through careful engineering of a new least-squares solver, which we call *Blendenpik*, and through extensive analysis and experimentation, we have been able to answer this question, yes.

Blendenpik beats LAPACK's direct dense least-squares solver by a large margin on essentially any dense tall matrix. Blendenpik is slower than LAPACK on tiny matrices, nearly square ones, and on some sparse matrices. But on a huge range of matrices of reasonable sizes, the answer is an unqualified yes. Figure 1.1 shows a preview of our experimental results. On large matrices, Blendenpik is about four times faster than LAPACK. We believe that these results show the potential of random-sampling algorithms and suggest that random-projection algorithms should be incorporated into future versions of LAPACK.

*Received by the editors August 12, 2009; accepted for publication (in revised form) January 6, 2010; published electronically April 23, 2010. This research was supported in part by an IBM Faculty Partnership Award and by grant 1045/09 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities).
http://www.siam.org/journals/sisc/32-3/76791.html

†Blavatnik School of Computer Science, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel-Aviv University, Tel-Aviv 69978, Israel (haima@post.tau.ac.il, stoledo@tau.ac.il).
‡Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139 (petar@csail.mit.edu).

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

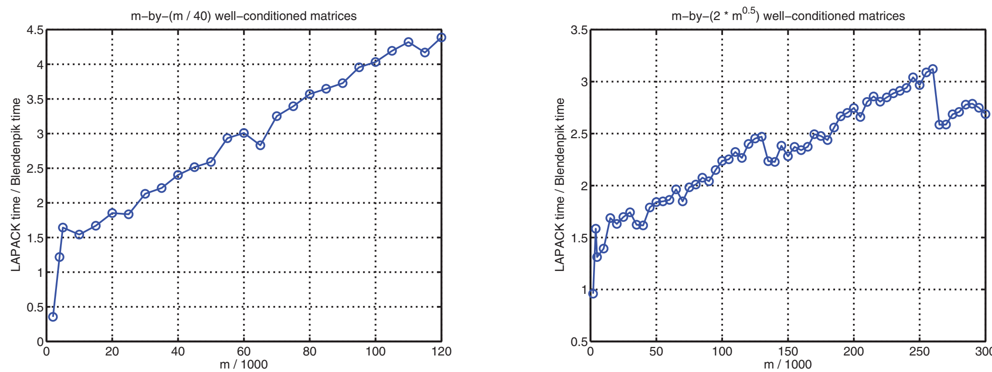

FIG. 1.1. *Comparison between* LAPACK *and the new solver for increasingly larger matrices. Graphs show the ratio of* LAPACK*'s running time to Blendenpik's running time on random matrices with two kinds of aspect ratios.*

2. Overview of the algorithm. Let $x_{\text{opt}} = \arg\min_x \|Ax - b\|_2$ be a large highly overdetermined system, with $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$. Can we sample a small set of rows, $\mathcal{R}$, and use only those rows to find an approximate solution? That is, is the solution $x_{\mathcal{R}} = \arg\min_x \|A_{\mathcal{R},*}x - b_{\mathcal{R}}\|_2$ a good approximation of x_{opt}? The following simple experiment in MATLAB [16] illustrates that for random matrices $x_{\mathcal{R}}$ is indeed a good approximation in some sense as long as $\mathcal{R}$ is big enough:

```
>> rand('state', 2378)
>> randn('state', 23984)
>> m = 20000; n = 100;
>> A = rand(m, n); b = rand(m, 1);
>> [U, S, V] = svd(A, 0);
>> S = diag(linspace(1, 10^6, 100));
>> A = U * S * V';
>> sampled_rows = find(rand(m, 1) < 10 * n * log(n) / m);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x = A \ b; >> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
     1.0084
```

The norm of the residual is within 1.01 of the optimal residual. In general, under certain conditions on A, a uniform random sample of $\Omega(n\log(m)\log(n\log(m)))$ rows leads to a residual that is within a factor of $1+\epsilon$ of the optimal with high probability [10]. These conditions hold in the experiment above, and the residual is indeed small. The paper [10] also proposes a more sophisticated algorithm that leads to a small residual for any matrix, but a small uniform sample does not work on any matrix.

There are two problems with this approach. First, the analysis in [10] bounds the relative error in residual norm, that is,

$$\|Ax_{\mathcal{R}} - b\|_2/\|Ax_{\text{opt}} - b\|_2 \le 1+\epsilon,$$

where x_{opt} is the true solution and $x_{\mathcal{R}}$ is the computed solution. Drineas et al. show that this implies a bound on the forward error,

$$\frac{\|x_{\text{opt}} - x_{\mathcal{R}}\|_2}{\|x_{\text{opt}}\|_2} \le \tan(\theta)\kappa(A)\sqrt{\epsilon},$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

where $\theta = \cos^{-1}(\|Ax_{\mathrm{opt}}\|_2/\|b\|_2)$. While such an analysis might be useful in some fields, it is difficult to relate it to standard stability analyses in numerical linear algebra. The standard stability analysis of least-squares algorithms is done in terms of *backward error*: an approximate solution $\tilde{x}$ is shown to be the exact solution of a perturbed system

$$\tilde{x} = \arg\min_x \|(A + \delta A)x - b\|_2 ,$$

where $\|\delta A\| \le \tilde{\epsilon}\|A\|$. This implies a bound on the forward error

$$\frac{\|x_{\mathrm{opt}} - \tilde{x}\|_2}{\|x_{\mathrm{opt}}\|_2} \le \left(\kappa(A) + \frac{\kappa(A)^2 \tan\theta}{\eta}\right)\tilde{\epsilon},$$

where $\eta = \|A\|_2\|x\|_2/\|Ax\|_2$. The two forward error bounds are not comparable in an obvious way. Moreover, the $\sqrt{\epsilon}$ appears to make it difficult to prove small forward error bounds in well-conditioned cases.

Second, running time depends on ϵ^{-1}. The backward stability requirement (e.g., value of ϵ) of linear algebra software may be a constant, but it is a tiny constant. So to achieve the required ϵ, the constants in the asymptotic bounds in [10] might be too large.

Rokhlin and Tygert [22] use a difference approach. They use the R factor of the sampled rows as a preconditioner in a Krylov-subspace method like LSQR [18]:

```
>> [Q, R] = qr(A1, 0);
>> x1 = lsqr(A, b, eps, 100, R);
lsqr converged at iteration 17 to a solution with relative
residual 0.5
```

A uniform sample of the rows is not always a good strategy. If A has a column j that is zero except for $A_{ij} \neq 0$, any subset of rows that excludes row i is rank deficient. If $m \gg n$, the algorithm needs to sample close to m rows in order to guarantee a high probability that row i is in the subset. If the row sample is too small, the preconditioner is rank deficient and LSQR fails.

```
>> A(1:end-1, end) = 0;
>> A1 = A(sampled_rows, :);
>> [Q, R] = qr(A1, 0);
>> x1 = lsqr(A, b, eps, 100, R);
Warning: Matrix is singular to working precision.
> In sparfun\private\iterapp at 33
  In lsqr at 217 In overview at 35
lsqr stopped at iteration 0 without converging to the desired
tolerance 2.2e-016 because the system involving the
preconditioner was ill conditioned.
The iterate returned (number 0) has relative residual 1
```

Uniform random sampling works well only when the *coherence* of the matrix is small, which is equal to the maximum norm of a row in Q, where Q forms an orthonormal basis for the column space of A (e.g., the leftmost factor in a reduced QR or singular-value decomposition; a formal definition of coherence appears in section 3). The coherence of the matrix is between n/m and 1. The lower it is, the better uniform sampling works.

```
>> [Q, R] = qr(A, 0);
>> coherence = max(sum(Q .^ 2, 2))
coherence =
    1.0000
```

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

The coherence of our matrix, after the insertion of zeros into column 1, is the worst possible.

The coherence of matrices with random independent uniform entries tends to be small, but as we have seen, other matrices have high coherence. We can use a randomized row-mixing preprocessing phase to reduce the coherence [10, 22]:

```
>> D = spdiags(sign(rand(m, 1)), 0, m, m);
>> B = dct(D * A); B(1, :) = B(1, :) / sqrt(2);
>> [Q, R] = qr(B, 0);
>> coherence = max(sum(Q .^ 2, 2))
coherence =
    0.0083
```

First, we randomly multiply each row by $+1$ or -1, and then apply a discrete cosine transform (DCT) to each column. The first row is divided by $\sqrt{2}$ to make the transformation orthogonal. With high probability the coherence of B is small. In the example above, it is less than twice the minimal coherence (0.005). There are many ways to mix rows to reduce the coherence. We discuss other methods in section 3.2.

With high probability, a uniform sample $B1$ of the rows of the row-mixed matrix B makes a good preconditioner. In the code below, we use the R factor of the sample to allow LSQR to apply the preconditioner efficiently:

```
>> B1 = B(sampled_rows, :);
>> [Q, R] = qr(B1, 0);
>> x1 = lsqr(A, b, eps, 100, R);
lsqr converged at iteration 15 to a solution with relative
residual 1
```

3. Theory.

This section explains the theory behind the algorithms that this paper investigates. The ideas themselves are not new; they have been proposed in several recent papers [10, 22, 17]. We do present some simple generalizations and improvements to existing results, but since the original proofs are strong enough for the generalizations, we omit the proofs, but they appear in [4].

3.1. Uniform sampling preconditioners.

The quality of uniform sampling preconditioners depends on how much the solution depends on specific rows. For example, if the sample is rank deficient unless row i is in it, then the size of a uniform sample must be too large to be effective. *Coherence* [6] is the key concept for measuring the dependence of the solution on specific rows.

DEFINITION 3.1. *Let A be an $m \times n$ full rank matrix, and let U be an $m \times n$ matrix whose columns form an orthonormal basis for the column space of A. The* coherence *of A is defined as*

$$\mu(A) = \max \|U_{i,*}\|_2^2 .$$

The coherence of a matrix is always smaller than 1 and bigger than n/m. If a row contains the only nonzero in one of the columns of A, then the coherence of the matrix is 1. Coherence does not relate in any way to the condition number of A.

Uniform random sampling yields a good preconditioner on incoherent matrices (matrices with small coherence). For example, if $\mu(A) = n/m$, then a sample of $\Theta(n \log n)$ rows is sufficient to obtain a good preconditioner. The following theorem describes a relationship between the coherence, the sample size, and the condition number of the preconditioned system.

THEOREM 3.2. *Let A be an $m \times n$ full rank matrix, and let $\mathcal{S}$ be a random sampling operator that samples $r \geq n$ rows from A uniformly. Let $\tau = C\sqrt{m\mu(A)\log(r)/r}$,*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

where C is some constant defined in the proof. Assume that $\delta^{-1}\tau < 1$. With probability of at least $1-\delta$, the sampled matrix $\mathcal{S}A$ is full rank, and if $\mathcal{S}A = QR$ is a reduced QR factorization of $\mathcal{S}A$, we have

$$\kappa(AR^{-1}) \leq \frac{1+\delta^{-1}\tau}{1-\delta^{-1}\tau}\,.$$

This result does not appear in this exact form in the literature, but its proof is a simple variation of the results in [10, 22]. Therefore, here we give only a sketch of the proof; the full version of the proof appears in [4]. The first phase is to bound $\|I_{n\times n}-(m/r)Q^T\mathcal{S}^T\mathcal{S}Q\|_2$ with high probability, using the proof technique of Lemma 5 from [10], in two steps. The first step bounds $\mathrm{E}(\|I_{n\times n}-(m/r)U^T\mathcal{S}^T\mathcal{S}U\|_2)$ using Lemma 4 from [10], and the second step uses Markov's inequality to bound $\|I_{n\times n}-(m/r)Q^T\mathcal{S}^T\mathcal{S}Q\|_2$ with high probability. Using a simple Rayleigh quotient argument, we then bound $\kappa(\mathcal{S}Q)$ with high probability. Finally, Theorem 1 in [22] shows that $\kappa(AR^{-1}) = \kappa(\mathcal{S}Q)$.

Remark 1. Notice that the condition number of the original matrix A does not affect the bound on the condition number of the preconditioned matrix.

Remark 2. Theorem 3.2 describes a relationship between sample size (r), the probability of failure (δ), and the condition number of the preconditioned system. With a small sample, the probability of obtaining a high condition number is high. A high condition number may lead to a large number of iterations in LSQR, but the number of iterations may also be small: the convergence of LSQR depends on the distribution of the singular values of AR^{-1}, not just on the extreme singular values. In fact, [3] uses the fact that a few very large or very small singular values do not affect convergence much.

If the coherence is high, uniform sampling produces poor preconditioners. One alternative is to use nonuniform sampling. Let $A = UR$ be a reduced QR factorization of A. Drineas, Mahoney, and Muthukrishnan [11] suggest sampling row i with probability $p_i = \|U_i\|_2^2/m$, where U_i is row i of U. Computing these probabilities requires too much work (a QR factorization of A), so to make this approach practical, probabilities should be somehow approximated; to the best of our knowledge, no efficient approximation algorithm has been developed yet. Therefore, in the next subsection we turn to a simpler approach, the one used by our solver, which is based on mixing rows.

3.2. Row mixing. Theorem 3.2 implies that even if there are important rows, that is, even if coherence is high, if we sample enough rows, then with high probability the preconditioner is a good preconditioner. The higher $\mu(A)$ is, the more rows should be sampled. This poses two problems. First, finding $\mu(A)$ is computationally hard. Second, if $\mu(A)$ is high too, then many rows need to be sampled. Indeed, if $\mu(A) = 1$ (the worst), then as many as $\mathcal{O}(m\log m)$ rows need to be sampled in order to get a bound on the condition number by using Theorem 3.2. When $\mu(A) = 1$, there is a row in the matrix that must be included in the sample for R to be full rank. We do not know which row it is, so no row can be excluded from the sample; this is not useful. If $\mu(A) = n/m$ (minimal coherence), on the other hand, then only $\Theta(n\log n)$ rows need to be sampled to get $\kappa = \mathcal{O}(1)$ with high probability.

In general, we cannot guarantee a bound on $\mu(A)$ in advance. The solution is to perform a *preprocessing* step in which rows are mixed so that their importance is nearly equal. The crucial observation is that a unitary transformation preserves the condition number but changes the coherence. If $\mathcal{F}$ is a unitary transformation and R is a preconditioner $\mathcal{F}A$, then R is an equally good preconditioner for A because

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

the singular values of AR^{-1} and $\mathcal{F}AR^{-1}$ are the same. But $\mu(A)$ and $\mu(\mathcal{F}A)$ are not necessarily the same; if we select $\mathcal{F}$ so that $\mu(\mathcal{F}A)$ is small, then we can construct a good preconditioner by uniformly random sampling the rows of $\mathcal{F}A$.

Any fixed unitary transformation $\mathcal{F}$ leads to a high $\mu(\mathcal{F}A)$ on some A's, so we use a random unitary transformation. We construct $\mathcal{F}$ from a product of a fixed *seed unitary transformation* F and a random diagonal matrix D with ± 1 diagonal entries. The diagonal entries of D are random, unbiased, independent random variables. The following theorem shows that with high probability, the coherence of $\mathcal{F}A$ is small, as long as the maximum value in F is not too large. It is a simple generalization of Lemma 3 in [10] using ideas from [17]; we omit the proof.

THEOREM 3.3. *Let A be an $m \times n$ full rank matrix, where $m \geq n$. Let F be an $m \times m$ unitary matrix, let D be a diagonal matrix whose diagonal entries are independent and identically distributed Rademacher random variables (*$\Pr(D_{ii} = \pm 1) = 1/2$*), and let $\mathcal{F} = FD$. With a probability of at least* 0.95*, we have*

$$\mu(\mathcal{F}A) \leq Cn\eta \log m \ ,$$

where $\eta = \max |F_{ij}|^2$ and some constant C.

Note 1. A must be full rank for the μ to be well defined. The theorem can be generalized to success guarantees other than 0.95. A higher probability leads to a higher constant C.

A seed matrix F is effective if it is easy to apply to A and if $\eta = \max |F_{ij}|^2$ is small. The minimal value of η is $1/m$. If η is $1/m$, then all the entries of F must have squared absolute values of $1/m$. A normalized DFT matrix has this property, and it can be applied quickly, but it involves complex numbers. A normalized Hadamard matrix has entries that are all $\pm 1/\sqrt{m}$, and in particular are all real. Hadamard matrices do not exist for all dimensions, but they do exist for powers of two, and they can be applied quickly at powers of two. The Walsh–Hadamard series

$$H_l = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \, H_{l+1} = \frac{1}{\sqrt{2}} \begin{pmatrix} H_l & H_l \\ H_l & -H_l \end{pmatrix} \ ,$$

enables the Walsh–Hadamard transform (WHT). Two other options for F are the discrete cosine transform (DCT) and discrete Hartley transform (DHT), which are real, exist for every size, and can be applied quickly. Their η value is $2/m$, twice as large as that of the WHT.

If we use one of the transformations described above, we need a sample of $\Theta(n \log(m) \log(n \log(m)))$ rows to obtain $\kappa = \mathcal{O}(1)$ with high probability. In practice, smaller samples are sufficient. In section 4, we discuss implementation issues and considerations for selecting the seed unitary transformation.

A possible alternative mixing strategy is a Kac random walk [14]. We define

$$\mathcal{F} = G_{T(m,n)} G_{T(m,n)-1} \cdots G_3 G_2 G_1,$$

where each G_t is a random Givens rotation. To construct G_t, we select two random indices i_t and j_t and a random angle θ_t, and we apply the corresponding Givens rotation. The number of rotations is chosen to make the coherence of $\mathcal{F}A$ sufficiently small with high probability. How small can we make $T(m,n)$? Ailon and Chazelle [1] conjecture that $T(m,n) = O(m \log m)$ will suffice, but they do not have a proof, so we do not currently use this approach. We propose an even simpler random walk, where instead of using a random angle θ_t, we fix $\theta_t = \pi/4$. We conjecture that still $T(m,n) = O(m \log m)$ will suffice, and we have verified this conjecture experimentally.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

3.3. High coherence due to a few rows. The coherence is the maximal row norm of U, an orthonormal basis for the column space of A. If all the rows of U have low coherence, a uniform random sample of the rows of A leads to a good preconditioner. We now show that even if a few rows in U have a large norm, a uniform random sample still leads to an effective preconditioner. The fundamental reason for this behavior is that a few rows with a large norm may allow a few singular values of the preconditioned system AR^{-1} to be very large, but the number of large singular values is bounded by the number of large rows. A few large singular vectors cause the condition number of AR^{-1} to become large, but they do not affect much the convergence of LSQR [3].

LEMMA 3.4. *Let A be an $m \times n$ full rank matrix, where $m \geq n$, and suppose we can write $A = \left[\begin{smallmatrix} A_1 \\ A_2 \end{smallmatrix}\right]$, where A_2 has $l \leq \min(m-n, n)$ rows. Let $\mathcal{S} \in \mathbb{R}^{k\times(m-l)}$ be a matrix such that $\mathcal{S}A_1$ is full rank. Let $\mathcal{S}A_1 = QR$ be the QR factorization of $\mathcal{S}A_1$. Then at least $n-l$ singular values of AR^{-1} are between the smallest singular value of A_1R^{-1} and the largest singular value of A_1R^{-1}.*

To prove Lemma 3.4 we need the following simplified version of Theorem 4.3 in [3].

THEOREM 3.5 (simplified version of Theorem 4.3 in [3]). *Let $A \in \mathbb{C}^{m\times n}$, and let $B \in \mathbb{C}^{k\times n}$ for some $1 \leq k < n$ be full rank matrices. Let $M \in \mathbb{C}^{n\times n}$ be a symmetric positive semidefinite matrix. If all the eigenvalues of (A^TA, M) are between α and β, then so are the $n-k$ smallest eigenvalues of $(A^TA + B^TB, M)$.*

Proof of Lemma 3.4. The singular values of A_1R^{-1} are the square root of the generalized eigenvalues of $(A_1^TA_1, (\mathcal{S}A_1)^T(\mathcal{S}A_1))$. The singular values of AR^{-1} are the square root of the generalized eigenvalues of $(A_1^TA_1 + A_2^TA_2, (\mathcal{S}A_1)^T(\mathcal{S}A_1))$. The matrix $A^TA = A_1^TA_1 + A_2^TA_2$ is an l-rank perturbation of $A_1^TA_1$, so according to Theorem 3.5 at least $n-l$ generalized eigenvalues of $(A_1^TA_1+A_2^TA_2, (\mathcal{S}A_1)^T(\mathcal{S}A_1))$ are between the smallest and largest generalized eigenvalues of $(A_1^TA_1, (\mathcal{S}A_1)^T(\mathcal{S}A_1))$. ∎

Suppose that A_1 is incoherent but A is coherent. In this case, coherency can be attributed to only a small number of rows (l rows). If A_1 is incoherent and full rank, then random sampling will produce a good preconditioner without row mixing. Lemma 3.4 implies that the same preconditioner will be a good preconditioner for A as long as l is small. In practice, we do not know the partition of A to A_1 and A_2. We simply sample from all the rows of A. But if m is large and the sample is small, the probability of missing any specific row is large; in particular, if l is small, then rows from A_2 are likely to be missed. The lemma shows that R is still a good preconditioner. If rows from A_2 are in the sample, the preconditioner is even better.

The lemma assumes that the row sample is full rank. In fact, almost the same result applies even if the sample is rank deficient, as long as we perturb R to make it full rank; see [3] for details.

4. Algorithm and implementation. In this section we summarize the three major steps of the algorithm: *row mixing (preprocessing)*, *row sampling and QR factorization*, and *iterative solution*. We also discuss how we handle random-sampling failures. The overall solver is presented in Algorithm 1.

Implementation. Our solver currently runs under MATLAB 7.7 [16], but it is implemented almost entirely in C. The C code is called from MATLAB using MATLAB's CMEX interface.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

ALGORITHM 1. BLENDENPIK'S ALGORITHM.

x=**blendenpik**($A \in \mathbb{R}^{m\times n}$,$b \in \mathbb{R}^n$)
▷ $m \geq n$, A is nonsingular
▷ parameters: γ and transform type

$$\tilde{m} \longleftarrow \begin{cases} 2^{\lceil \log_2 m \rceil}, & \text{WHT} \\ \lceil m/1000 \rceil \times 1000, & \text{DCT or DHT} \end{cases}$$

$$M \longleftarrow \begin{bmatrix} A \\ 0 \end{bmatrix} \in \mathbb{R}^{\tilde{m}\times n}$$

while not returned
  $M \longleftarrow F_{\tilde{m}}(DM)$
    ▷ D is a diagonal matrix with ± 1 on its diagonal with equal probability
    ▷ $F_{\tilde{m}}$ is the seed unitary transform (WHT/DCT/DHT), $\Theta(mn \log m)$ operations
  Let $\mathcal{S} \in \mathbb{R}^{\tilde{m}\times\tilde{m}}$ be a random diagonal matrix:

$$\mathcal{S}_{ii} = \begin{cases} 1 & \text{with probability } \gamma n/\tilde{m} \\ 0 & \text{with probability } 1 - \gamma n/\tilde{m} \end{cases}$$

  Factorize: $\mathcal{S}M = QR$, reduced QR factorization ($R \in \mathbb{R}^{n\times n}$)
  $\tilde{\kappa} \longleftarrow \kappa_{\text{estimate}}(R)$, condition number estimation (LAPACK's DTRCON)
  if $\tilde{\kappa}^{-1} > 5\epsilon_{\text{machine}}$
    $x \longleftarrow \text{LSQR}(A, b, R, 10^{-14})$
    return
  else
    if #iterations > 3
      failure: solve using LAPACK and return
    end if
  end if
end while

Row mixing. In section 3.2 we suggest five row-mixing strategies: DFT, DCT, DHT, WHT, and Kac. We chose not to implement DFT and Kac. The DFT of a vector is a complex vector even if the vector is real. Thus, using DFT entails operation-count and memory penalties on subsequent phases when applied on real matrices. Therefore, it is unlikely that an FFT-based algorithm would outperform one based on DCT or DHT. Kac's random walk appears to suffer from poor cache locality due to random index selection.

WHT is theoretically optimal, in the sense that its η value is $1/m$, but it can be applied only if the number of rows is a power of two. By padding the matrix with zeros we can apply WHT to smaller matrices. This causes discontinuous increases in the running time and memory usage as m grows. We use SPIRAL WHT [13] to apply

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

WHT. To get good performance it is essential to use the package's self-optimization feature, which incurs a small one-time overhead.

Instead of using WHT, any Hadamard matrix can be used. If H_1 and H_2 are Hadamard matrices, then so is $H_1 \otimes H_2$, so by using kernels of small Hadamard transforms, efficient large Hadamard transforms can be implemented. But to the best of our knowledge, there is currently no efficient implementation of this idea.

DCT and DHT are near-optimal alternatives (their η value is $2/m$). Their advantages over WHT are that they exist for all vector sizes and that, in principle, they can always be applied in $O(m \log m)$ operations. However, in practice these transforms are quite slow for some sizes. The performance of fast transforms (DCT and DHT) depends on how the input size m can be factored into integers. The performance is not monotone in m. Also, the fast-transform library that we use (FFTW) requires tuning for each input size; the tuning step also takes time. To address these issues, we use the following strategy. During the installation of our solver, we generate tuned DCT and DHT solvers for sizes of the form $m = 1000k$, where k is an integer. The information used by FFTW to generate the tuned solvers (called "wisdom" in FFTW) is kept in a file. Before the solver uses FFTW to compute DHT or DCT, this information is loaded into FFTW, so no additional tuning is done at solve time. Before applying DCT or DHT to a matrix, we pad the matrix to the next multiple of 1000 or to a slightly higher multiple if the tuning step suggested that the higher multiple would result in higher performance. One can imagine more sophisticated strategies, based on knowing what kernel sizes FFTW uses as fast building blocks and using sizes that are multiples of those building blocks. The method that we used is not optimal, but it does deliver good performance while keeping tuning time reasonable.

We tune FFTW using aggressive settings, so tuning takes a long time (hours). We also experimented with milder tuning settings. If FFTW's weakest tuning is used, the tuning time of DHT reduces to about 11 minutes, but the time spent in computing the DHTs is sometimes doubled. As we shall see in section 5.6, this slows our solver, relative to aggressive setting, by at most 15% (usually less).

Sampling rows and *QR* factorization. We sample rows by generating a size $\tilde{m}$ vector with random uniform entries in $[0, 1]$, where $\tilde{m}$ is the number of rows after padding. We use MATLAB's `rand` function to generate the vector. A row is sampled if the corresponding entry in the vector is smaller than $\gamma n/\tilde{m}$, where γ is a parameter. The expected number of rows that are sampled is γn, but the actual value can be higher or smaller. This is the same strategy that was suggested in [10]. Once the rows are sampled we compute their QR factorization using LAPACK's DGEQRF function.

Row sampling can be combined with row mixing to improve the asymptotic running time. Any k indices of the FFT of a m element vector can be computed using only $\mathcal{O}(m \log k)$ operations [24]. This is also true for WHT [2]. If we select the sampled rows before the row mixing, we can compute only the mixed rows that are in the sample. We do not use this strategy because the libraries that we use do not have this option.

Iterative solution. We use LSQR to find the solution. Given an iterate x_j with a corresponding residual $r_j = b - Ax_j$, stopping the algorithm when

$$\frac{\|A^T r_j\|_2}{\|A\|_F \|r_j\|_2} \leq \rho \tag{4.1}$$

guarantees that x_j is an exact solution of

$$x_j = \arg\min_x \|(A + \delta A)x - b\|_2$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

where $\|\delta A\|_F \leq \rho \|A\|_F$. That is, the solution is backward stable [7]. The value of ρ is a parameter that controls the stability of the algorithm. To use this stopping criterion, we need to compute r_j and $A^T r_j$ in every iteration. It is therefore standard practice in LSQR codes to estimate $\|r_j\|_2$ and $\|A^T r_j\|_2$ instead of actually computing them. The estimate formulas used are accurate in exact arithmetic, and in practice they are remarkably reliable [18]. If a preconditioner R is used, as in our algorithm, $\|A^T r_j\|_2$ cannot be estimated but $\|(AR^{-1})^T r_j\|_2$ can be. Preconditioned LSQR codes estimate $\|AR^{-1}\|_F$ as well and use the stopping criterion

$$\frac{\| \left(AR^{-1}\right)^T r_j\|_2}{\|AR^{-1}\|_F \|r_j\|_2} \leq \rho$$

that guarantees a backward stable solution to

$$y_j = \arg\min_x \left\|AR^{-1}y - b\right\|_2$$

and returns $x_j = R^{-1}y_j$. We use the same strategy in our solver. We set $\rho = 10^{-14}$, which is close to $\epsilon_{\text{machine}}$ but not close enough to risk stagnation of LSQR. This setting results in a solver that is about as stable as a QR-based solver.

Most of the LSQR running time is spent on multiplying vectors by A and A^T. If A is sparse and very tall, using a sparse matrix-vector multiplication code can reduce the LSQR running time, even though R is dense. We have not exploited this opportunity in our code.

Handling failures. The bounds in section 3 hold with some probability bounded from below. With some probability, the algorithm can fail to produce an effective preconditioner in one of two ways: (1) the preconditioner can be rank deficient or highly ill conditioned, or (2) the condition number $\kappa(AR^{-1})$ can be high. When the condition number is high, LSQR converges slowly, but the overall algorithm does not fail. But a rank-deficient preconditioner cannot be used with LSQR. To address this issue, we estimate the condition number of the preconditioner R using LAPACK's DTRCON function. If the condition number is too high (larger than $\epsilon_{\text{machine}}^{-1}/5$), we perform another row-mixing phase and resample. If we repeat this three times and still do not get a full rank preconditioner, we give up, assume that the matrix itself is rank deficient, and use LAPACK. This never happened in our experiments on full rank matrices, but on some matrices we had to mix and sample more than once.

5. Numerical experiments.

We experimented with the new algorithm extensively in order to explore its behaviors and to understand its performance. This section reports the results of these experiments (Figure 1.1 shows additional results).

5.1. Experimental setup.

We compare the new solver, which we call *Blendenpik*, to a high-performance dense QR solver and to LSQR with no preconditioning. The dense QR solver is LAPACK's DGELS: a high-performance, high-quality, portable code. We call LAPACK from MATLAB using a special CMEX interface that measures only LAPACK's running time. No MATLAB-related overheads are included; MATLAB is used here only as a scripting tool.

Running times were measured on a machine with two AMD Opteron 242 processors (we used only one) running at 1.6 GHz with 8 GB of memory. We use GOTO BLAS 1.30 and LAPACK 3.2.1 for basic matrix operations and FFTW 3.2.1 for the DCT and DHT.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

The measured running times are wall-clock times that were measured using the `ftime` Linux system call.

We evaluated our solver on several classes of random matrices. Random matrices were generated using MATLAB's `rand` function (random independent uniform numbers). Ill-conditioned matrices are obtained by generating their SVD decomposition: two random orthonormal matrices and an equally spaced diagonal matrix with the appropriate condition number.

Our solver relies on automatic tuning of the fast-transform libraries that it uses (FFTW and SPIRAL). This is an installation-time overhead that is not included in our running-time measurements. Automatic tuning is a technique of growing importance in various numerical libraries, such as the ATLAS [26] implementation of the BLAS.

Theoretical bounds relate to the coherence, which is the maximum row norm in the orthogonal factor of the matrix. Our experiments suggest that, in practice, running time is related to the number of rows that have a large norm in the orthogonal factor. Therefore, we experimented with three types of matrices: *incoherent* matrices, *semicoherent* matrices and *coherent* matrices. Incoherent matrices $X_{m\times n}$, either well conditioned or ill conditioned, are generated using the `rand` function with no restriction on the structure. Semicoherent matrices, are of the form

$$Y_{m\times n} = \begin{bmatrix} \tilde{B} & \\ & I_{n/2} \end{bmatrix} + 10^{-8} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & & \vdots \\ 1 & \cdots & 1 \end{bmatrix},$$

where $\tilde{B}$ is an $(m-n/2)\times n/2$ rectangular random matrix and $I_{n/2}$ is a square identity of dimension $n/2$. $Y_{m\times n}$ is, in fact, coherent ($\mu(Y_{m\times n}) = 1$), but only $n/2$ rows have a large norm in the orthogonal factor. Our coherent matrices have the form

$$Z_{m\times n} = \begin{bmatrix} D_{n\times n} \\ 0_{(m-n)\times n} \end{bmatrix} + 10^{-8} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & & \vdots \\ 1 & \cdots & 1 \end{bmatrix},$$

where $D_{n\times n}$ is a random diagonal matrix. The orthogonal factors of these matrices have n rows with a large norm. In both semicoherent and coherent matrices, the constant 10^{-8} matrix is added to make the matrices dense. Some solvers, including LAPACK's (in version 3.2.1), exploit sparsity. Our solver does not. We added the constant matrix to avoid this source of variance; we acknowledge the fact that, for some sparse matrices, LAPACK's dense solver is faster than our solver.

5.2. Tuning experiments. The behavior of our solver depends on the seed unitary transformation that mixes the rows, on the number of row-mixing steps, and on the sample size. These parameters interact in complex ways, which are not fully captured by asymptotic analyses. We begin with experiments that are designed to help us choose these parameters in the algorithm.

5.2.1. Unitary transformation type. The row-mixing phase uses a fixed seed unitary matrix that depends only on the row dimension of the problem. In 3.2 we suggested five different seed unitary matrices. As explained in section 4, we implemented only three of them, all using external libraries: the WHT, the DCT, and the DHT. Figures 5.1 shows the running time of each transformation time on increasingly larger matrices. WHT is the fastest, but DHT and DCT come close.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

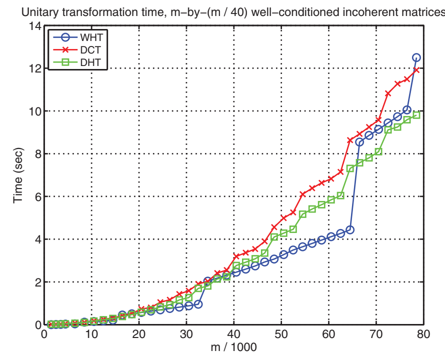

FIG. 5.1. *Time spent on the fast unitary transformation (row mixing) for increasingly larger matrices. We tested all three implemented transforms: WHT, DCT, and DHT.*

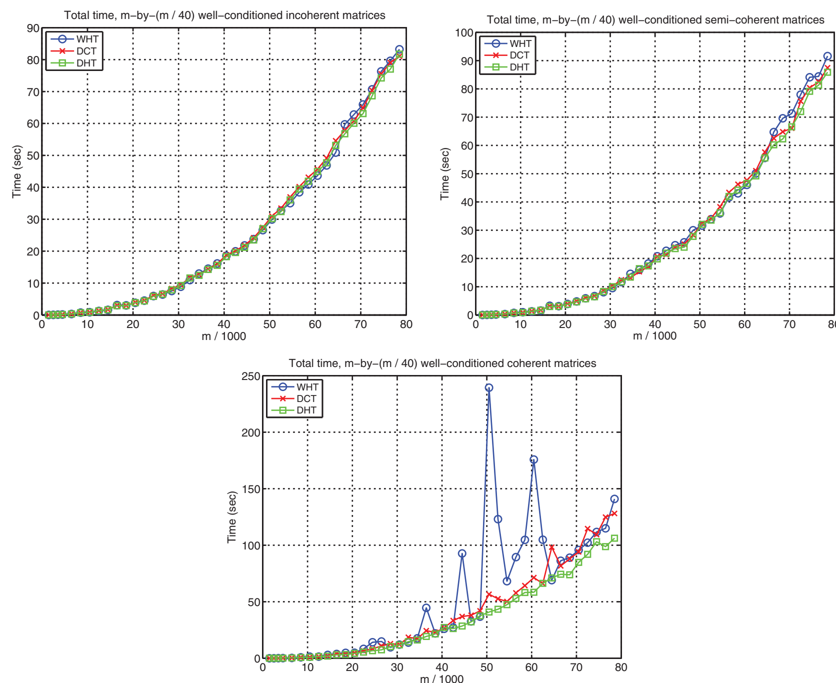

FIG. 5.2. *Overall running time of the algorithm with different fast unitary transforms (row mixing) on increasingly larger matrices. We tested on incoherent matrices (top left graph), semicoherent matrices (top right graph), and coherent matrices (bottom graph).*

Different unitary transforms improve coherence in different ways. Figure 5.2 examines the overall running time of the solver on incoherent, semicoherent, and coherent matrices. For incoherent and semicoherent matrices there does not seem to be a significant difference between the different mixing methods. WHT's overall time is smaller because it is faster than other methods. On coherent matrices, WHT exhibits poor and erratic performance. Usually, a single WHT phase generated a very ill-conditioned preconditioner (very close to rank deficiency). This was sometimes detected by the condition number estimator, in which case a second WHT phase was

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

done. In some cases the condition number estimator test failed, and convergence was very slow. DHT and DCT continue to work well on coherent matrices; the two methods behave the same. It is interesting to note that, from a theoretical standpoint, WHT is superior, but in practice, DHT and DCT work better.

Clearly, WHT's advantage (fast application and a low η) are offset by its disadvantages (reduced robustness and a large memory footprint). We therefore decided to use DHT (which is faster than DCT) for all subsequent experiments except for the right graph in Figure 1.1, where we used WHT for experimental reasons.

5.2.2. Sample size and number of row-mixing steps. The theoretical analysis shows that sampling $\Omega(n \log(m) \log(n \log(m)))$ rows is sufficient with high probability, but we do not know the constants in the asymptotic notation. The analysis may give bounds in the probability of failure, but even if there is failure (e.g., the condition number is bigger than the bound), running time might still be good. Convergence behavior is governed by the distribution of singular values, and it is not fully captured by the condition number. The contributions of each phase to the running time interact in a complex way that is not fully predictable by worst-case asymptotic analysis. Therefore, we performed experiments whose goal is to determine a reasonable sampling size.

We also need to decide on the number of row-mixing steps. Row-mixing steps reduce the coherence and improve the outcome of random sampling. Theoretical bounds state that after a single row-mixing step, the coherence is within a $O(\log m)$ factor of the optimal with high probability. Therefore, after the first row-mixing step, there is still room for improvement. Although there are no theoretical results that state so, it reasonable to assume that additional row-mixing steps will reduce coherence further, which will cause LSQR to converge faster, perhaps offsetting the cost of the extra mixing steps.

Figure 5.3 presents the results of these experiments. We ran experiments with two matrix sizes, $30{,}000 \times 750$ (left graphs) and $40{,}000 \times 2{,}000$ (right graphs), and all matrix types, incoherent (top left and middle right graphs), semicoherent (top right and bottom left graphs), and coherent (middle left and bottom right graphs). All the matrices were ill conditioned.

We used sample size γn, where γ ranges from 1.5 to 10. Although the theoretical bound is superlinear, it is not necessarily tight. As the results show, for the range of matrices tested in our experiments, the best sample size displays a *sub*linear (in n) behavior (which might change for larger matrices).

For $30{,}000 \times 750$ matrices the best sample size is around $\gamma = 6$. For $40{,}000 \times 2{,}000$ it is $\gamma = 3$. Apparently, for larger matrices a smaller sample is needed (relative to n), contrary to the theoretical analysis. A sample size with $\gamma = 4$ is close to optimal for all matrices. For incoherent and semicoherent matrices there is a (small) advantage for using only one preprocessing phase. For coherent matrices the best results are achieved when using two preprocessing phases. In fact, using only one preprocessing phase can be disastrous when combined with a sample size that is too small. But with a sample size $\gamma = 4$, near-optimal results can be achieved with only one preprocessing phase.

Following these experiments we decided to fix $\gamma = 4$ and to use one preprocessing phase. We used these settings for the rest of the experiments. These parameters are not optimal in all cases, but they seem to be nearly optimal for most cases. The rest of the experiments in this paper use these values.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

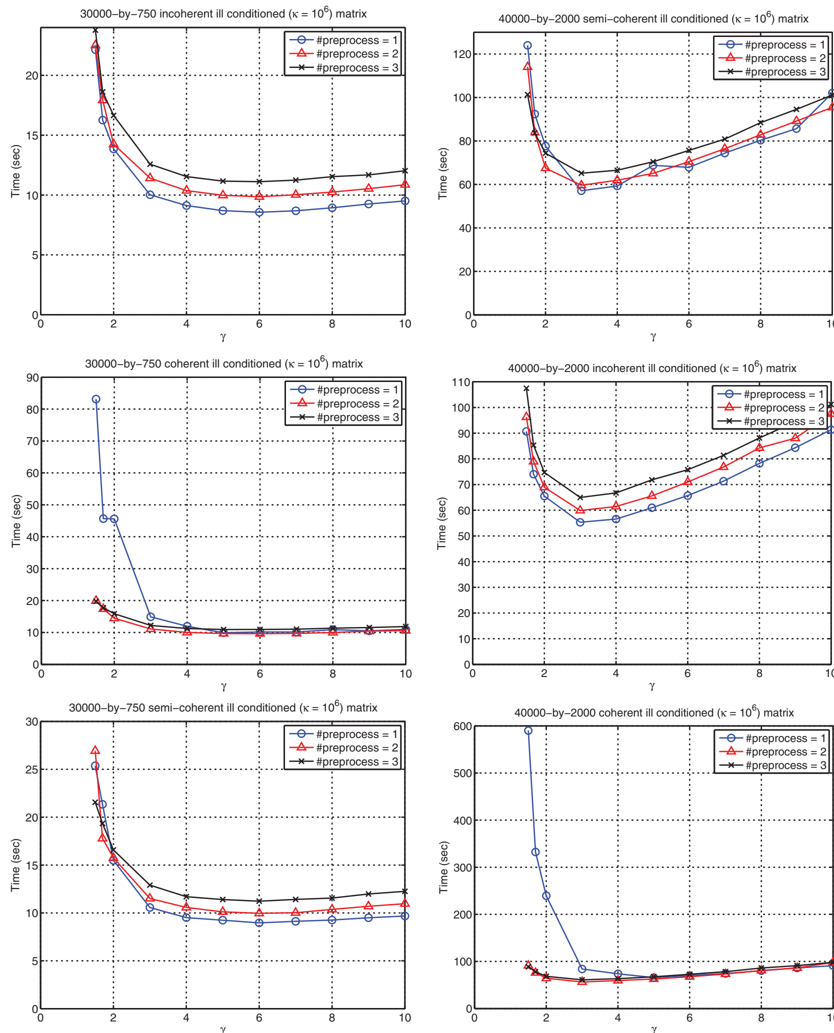

FIG. 5.3. *Running time as a function of sample size and number of row-mixing steps for* $30{,}000 \times 750$ *matrices (left graphs) and* $40{,}000 \times 2{,}000$ *matrices (right graphs). We ran the same experiment on incoherent matrices (top left and middle right graphs), semicoherent matrices (top right and bottom left graph), and coherent matrices (middle left and bottom right graphs).*

5.3. Ill-conditioned matrices. Figure 5.4 shows that the condition number of A does not affect our new solver at all, but it does affect unpreconditioned LSQR. On very well conditioned matrices, unpreconditioned LSQR is faster, but its performance deteriorates quickly as the condition number grows.

5.4. Easy and hard cases. Figure 5.5 compares the performance of our solver and of LAPACK on incoherent, semicoherent, and coherent matrices of four different aspect ratios. The number of elements in all matrices is the same. (LAPACK's running time depends only on the matrix's dimensions, not on its coherence, so the graph shows only one LAPACK running time for each size.) Our solver is slower on matrices with high coherence than on matrices with low coherence but not by much. Even when the coherence is high, our solver is considerably faster than LAPACK. Hard cases (high coherence) run slower because LSQR converges slower, so more LSQR iterations are

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

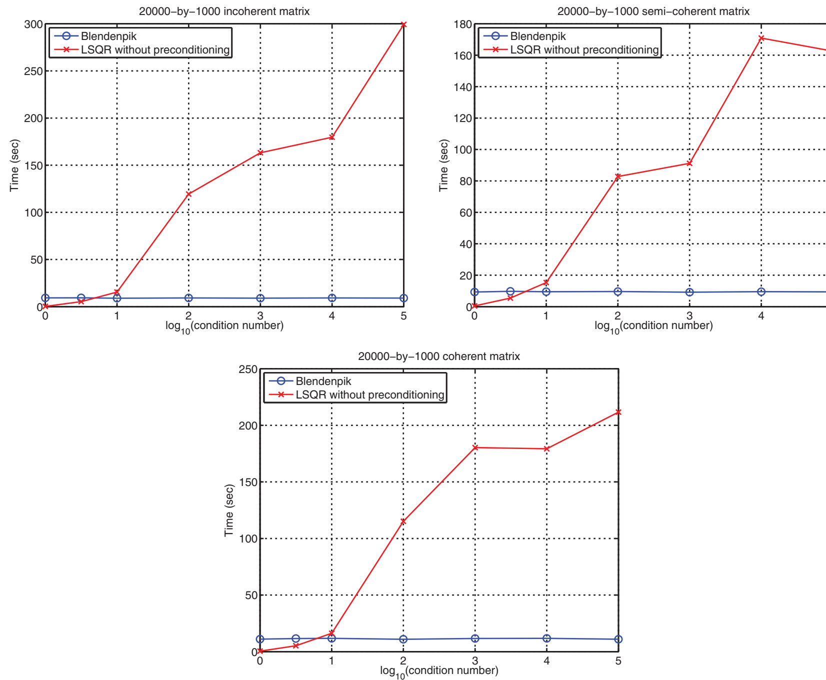

FIG. 5.4. *Running time on increasingly ill-conditioned matrices.*

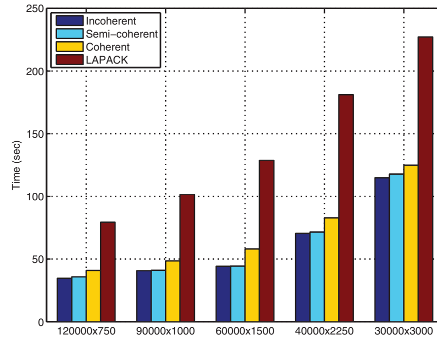

FIG. 5.5. *Running time on different coherence profiles.*

performed. (The other phases of the algorithm are oblivious to coherence.) It appears that a single row-mixing phase does not remove the coherence completely.

5.5. Convergence rate. In the experiments whose results are shown in the left graph in Figure 5.6, we examine the LSQR convergence rate on a single matrix. The graph shows the norm of the residual after each iteration. Except for the final iterations, where the solver stagnates near convergence, the convergence rate is stable and predictable. This is a useful property that allows us to predict when the solver will converge and to predict how the convergence threshold affects the running time. The rate itself is slower on coherent matrices than on incoherent and semicoherent ones. This is the same issue we saw in Figure 5.5.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

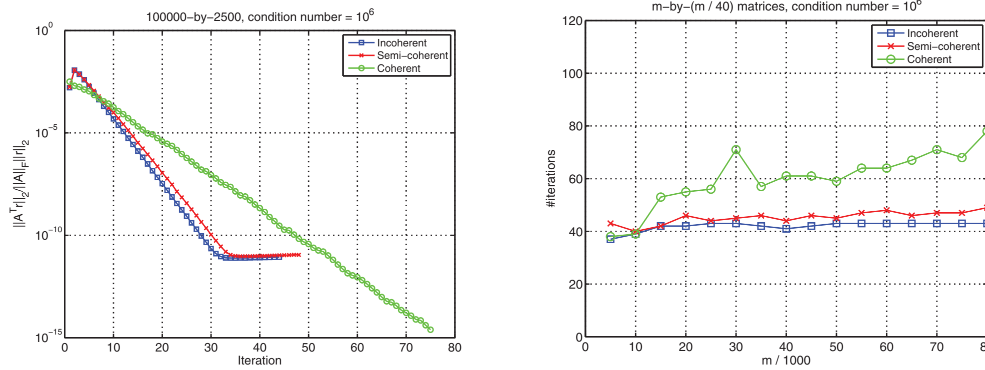

FIG. 5.6. *Convergence rate experiments. The left graph shows* $\|A^T r^{(i)}\|_2/\|A\|_F\|r^{(i)}\|_2$*, where* $r^{(i)}$ *is the residual after the ith iteration of LSQR on three* 100,000 × 2,500 *matrices of different coherence profiles. The right graph shows the number of LSQR iterations needed for convergence on increasingly larger matrices.*

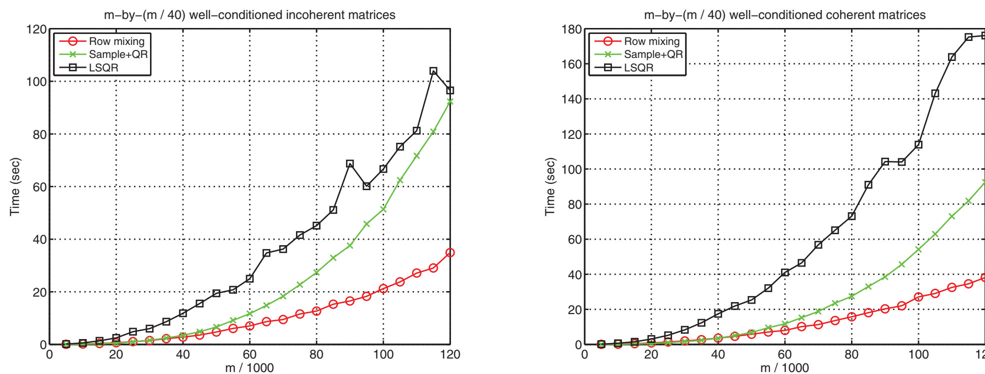

FIG. 5.7. *Breakdown of running time on increasingly larger matrices. The plotted series shows the running time of each phase. The left graph shows the breakdown for incoherent matrices, while the right graph shows the breakdown for coherent matrices.*

The graph on the right examines the number of iterations required for LSQR to converge as a function of problem size. On incoherent and semicoherent matrices the number of iterations grows very slowly. On coherent matrices the number of iterations grows faster.

5.6. The cost of the different phases. Figure 5.7 shows a breakdown of the running time of our solver for incoherent matrices (left graph) and coherent matrices (right graph) of increasingly larger size. The row-mixing preprocessing phase is not a bottleneck of the algorithm. Most of the time is spent on factoring the preconditioner and on LSQR iterations. The most expensive phase is the LSQR phase. The asymptotic running time of the row-mixing phase is $\Theta(mn \log m)$, and for the QR phase it is $\Theta(n^3)$. Each LSQR iteration takes $\Theta(mn)$ time, and the number of iterations grows slowly. In both graphs $n = m/40$, so the QR phase is asymptotically the most expensive.

The dominance of the LSQR phase implies that considerable speedup can be achieved by relaxing the convergence threshold. In our experiments the convergence threshold was set to 10^{-14}. If a convergence threshold of 10^{-6} is acceptable, for

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

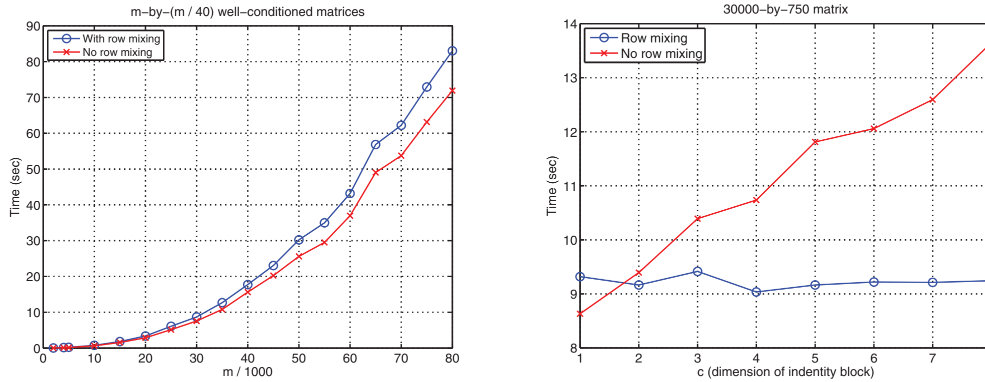

FIG. 5.8. *Experiments examining strategies with no row mixing versus the regular strategy. The left graph compares the solver without a row-mixing phase to the solver with a row-mixing phase on incoherent matrices. The right graph compares the same two solvers on matrices with a few important rows.*

example, we can roughly halve the number of iterations of the LSQR phase, thereby accelerating our solver considerably.

The row mixing phase takes about 15% of overall solver time. Even if we double the row-mixing time, our solver will still be faster than LAPACK on nearly all of the matrices used in our experiments.

5.7. No row mixing. If a matrix is completely incoherent to begin with, we do not need to mix its rows. On such matrices, row mixing takes time but does not reduce the running time of subsequent phases. The left graph in Figure 5.8 shows that this is essentially true on random matrices, which have low (but not minimal) coherence; the algorithm runs faster without mixing at all.

The right graph in Figure 5.8 examines performance on coherent matrices whose coherence can be attributed to a small number c of rows. The matrices are of the form

$$S_{(m+c)\times(n+c)} = \begin{bmatrix} S_0 & S_1 \\ 0 & 10^3 \times I_c \end{bmatrix},$$

where $S_0 \in \mathbb{R}^{m\times(n-c)}$and $S_1 \in \mathbb{R}^{m\times c}$ are a random rectangular matrix and I_c is a c-by-c identity. When c is tiny (1 and 2), row mixing does not improve the running time substantially. But for $c > 2$, with row mixing the running time remains constantly low, while a performance of random sampling without mixing deteriorates as the size of the $10^3 \times I_c$ block grows.

The reason for the deterioration is numerical inaccuracy and not poor preconditioning. The basis vectors generated by LSQR lose orthogonality because a short recurrence (Lanczos recurrence) is used. A celebrated result of Paige [19] shows that loss of orthogonality is large only in the directions of converged or nearly converged Ritz vectors. As long as no Ritz has converged, a satisfactory level of orthogonality is maintained. This result explains why isolated singular values in the preconditioned matrix cause numerical problems: the algorithm tends to converge fast for the isolated eigenvalues. Possible solutions for this problem are full orthogonalization (expensive), selective orthogonalization [20], and others (see section 5.3 in [25]). We have verified this observation by running LSQR with full orthogonanlization (graph not included).

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

6. Discussion and related work. Experiments show that our solver is faster than LAPACK and faster than LSQR with no preconditioning. The algorithm is robust and predictable. The algorithm is competitive in the usual metric of numerical linear algebra, and it demonstrates that randomized algorithms can be effective in numerical linear algebra software. We have not encountered cases of large dense matrices where the solver fails to beat LAPACK, even in hard test cases, and we have not encountered large variance in running time of the algorithm on a given matrix. Even the convergence rate in the iterative phase is stable and predictable (unlike many algorithms that use an iterative method).

Although, the numerical experiments demonstrate the validity of the worst-case theoretical analysis, they also demonstrate that actual performance is not fully described by it. In some issues actual performance acts differently than suggested by theoretical bounds, or the observed behavior is not explained by the analysis:

- The theoretical analysis suggests that WHT is better in reducing coherence. In practice DHT and DCT work better, even though it takes longer to compute them. In fact, on highly coherent matrices, WHT sometimes fails to mix rows well enough (so we need to apply it again), while this never happened for DHT and DCT.
- The algorithm may fail with some small probability. It may fail to produce an incoherent matrix after row sampling, and important rows may be left out of the random sample (thereby producing a poor preconditioner). Some failures may slow down the solver considerably (for example, when the preconditioner is rank deficient and another row-mixing phase is necessary), but it is practically impossible for the algorithm not to finish in finite time on full rank matrices. Current theory does not guarantee that the probability of slowdown is negligible. When using WHT for row mixing, the solver did slow down sometimes due to such failures. When DHT is used for row mixing, we did not encounter such failures, and running time was always good, with a small variance. Apparently the actual probability of failure is much smaller than the theoretical bounds.
- Theoretical bounds require a superlinear sample size. In practice, a linear sample works better. It is unclear whether the reason is that the bounds are not tight or whether constants come into play.
- The theory relates performance to the coherence of the matrix. Coherence uses the maximum function, which from our experiment is too crude for analyzing random sampling. Actual performance depends on the distribution of row norms in the orthogonal factor, not just the maximum values. In a sense, the role coherence is similar to the role of the condition number in Krylov methods: it provides bounds using extreme values (easy to handle) while actual performance depends on internal values (hard to handle).

The algorithm used by our solver is new, but its building blocks are not. We chose building blocks that are geared toward an efficient implementation. Using WHT for row mixing (and padding the matrix by zeros) was suggested by Drineas et al. [10]. Their complete method is not suitable for a general-purpose solver because sample size depends on the required accuracy. Using DCT or DHT for row mixing in low rank matrix approximations was suggested by Nguyen, Do, and Tran [17]. Their observation carries to a least-squares solution. DHT has a smaller memory footprint than WHT, and it works better than WHT and DCT, so we decided to use it. Using the sampled matrix as a preconditioner for an iterative Krylov-subspace method was suggested by Rokhlin and Tygert [22]. They use CGLS; we decided to use LSQR

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

because it often works better. The row-mixing method in [22] uses FFT, which forces the solver to work on complex numbers. Furthermore, their analysis requires two FFT applications.

Our observation that the solver can work well even if the post-mixing coherence is high, as long as the number of high-norm rows in U is small, is new.

Unlike previous work in this area, we compared our solver to a state-of-the-art direct solver (LAPACK), showed that it is faster, and explored its behavior on a wide range of matrices. Drineas et al. [10] do not implement their algorithm. Rokhlin and Tygert [22] implemented their algorithm, but they compared it to a direct solver that they implemented, which is probably slower than LAPACK's. They also experimented with only a small range of matrices (incoherent matrices whose number of rows is a power of two).

Acknowledgments. This research was motivated by discussions with Michael Mahoney concerning the theoretical analysis of random sampling algorithms for least-squares regression. We thank Mike for these discussions, and we thank Schloss Dagstuhl for making them possible (as part of the workshop on Combinatorial Scientific Computing). It is also a pleasure to thank Vladimir Rokhlin and Mark Tygert for illuminating discussions of their results. We thank the referees for valuable comments and suggestions.

REFERENCES

[1] N. Ailon and B. Chazelle, *Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform*, in Proceedings of the 38th Annual ACM Symposium on Theory of Computing, New York, 2006, pp. 557–563.

[2] N. Ailon and E. Liberty, *Fast dimension reduction using Rademacher series on dual BCH codes*, in Proceedings of the 19th Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 2008, pp. 1–9.

[3] H. Avron, E. Ng, and S. Toledo, *Using perturbed QR factorizations to solve linear least-squares problems*, SIAM J. Matrix Anal. Appl., 31 (2009), pp. 674–693.

[4] H. Avron, *Efficient and Robust Hybrid Iterative-Direct Multipurpose Linear Solvers*, Ph.D. thesis, Tel-Aviv University, Israel, expected October 2010.

[5] C. Boutsidis and P. Drineas, *Random projections for the nonnegative least-squares problem*, Linear Algebra Appl., 431 (2009), pp. 760–771.

[6] E. J. Candès and B. Recht, *Exact matrix completion via convex optimization*, Tecnical report CoRR, abs/0805.4471 (2008).

[7] X.-W. Chang, C. C. Paige, and D. Titley-Peloquin, *Stopping criteria for the iterative solution of linear least squares problems*, SIAM J. Matrix Anal. Appl., 31 (2009), pp. 831–852.

[8] A. Dasgupta, P. Drineas, B. Harb, R. Kumar, and M. W. Mahoney, *Sampling algorithms and coresets for ℓ_p regression*, SIAM J. Comput., 38 (2009), pp. 2060–2078.

[9] J. Demmel, I. Dumitriu, and O. Holtz, *Fast linear algebra is stable*, Numer. Math., 108 (2007), pp. 59–91.

[10] P. Drineas, M. W. Mahoney, S. Muthukrishnan, and T. Sarlós, *Faster least squares approximation*, Tecnical report, CoRR, abs/0710.1435 (2007).

[11] P. Drineas, M. W. Mahoney, and S. Muthukrishnan, *Sampling algorithms for l2 regression and applications*, in Proceedings of the 17th Annual ACM–SIAM Symposium on Discrete Algorithms, ACM, New York, 2006, pp. 1127–1136.

[12] P. Drineas, M. W. Mahoney, and S. Muthukrishnan, *Relative-error CUR matrix decompositions*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 844–881.

[13] J. Johnson and M. Puschel, *In search of the optimal Walsh-Hadamard transform*, in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Washington, DC, 2000, pp. 3347–3350.

[14] M. Kac, *Probability and Related Topics in Physical Science*, Wiley Interscience, New York, 1959.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

[15] M. W. Mahoney and P. Drineas, *CUR matrix decompositions for improved data analysis*, Proc. Nat. Acad. Sci., USA, 106 (2009), pp. 697–702.
[16] MATLAB, version 7.7, software package, The MathWorks, 2008.
[17] N. H. Nguyen, T. T. Do, and T. D. Tran, *A fast and efficient algorithm for low-rank approximation of a matrix*, in Proceedings of the 41st ACM Symposium on Theory of Computing, New York, 2009, pp. 215–224.
[18] C. C. Paige and M. A. Saunders, *LSQR: An algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Software, 8 (1982), pp. 43–71.
[19] C. C. Paige, *The Computation of Eigenvalues and Eigenvectors of Very Large Sparse Matrices*, Ph.D. thesis, University of London, 1971.
[20] B. N. Parlett and D. S. Scott, *The Lanczos algorithm with selective orthogonalization*, Math. Comp., 33 (1979), pp. 217–238.
[21] V. Rokhlin, A. Szlam, and M. Tygert, *A randomized algorithm for principal component analysis*, SIAM J. Matrix Anal. Appl., 31 (2009), pp. 1100–1124.
[22] V. Rokhlin and M. Tygert, *A fast randomized algorithm for overdetermined linear least-squares regression*, Proc. Nat. Acad. Sci., USA, 105 (2008), pp. 13212–13217.
[23] T. Sarlos, *Improved approximation algorithms for large matrices via random projections*, in Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, Washington, DC, 2006, pp. 143–152.
[24] H. V. Sorensen and C. S. Burrus, *Efficient computation of the DFT with only a subset of input or output points*, IEEE Trans. Signal Proces., 41 (1993), pp. 1184–1200.
[25] G. W. Stewart, *Matrix Algorithms Volume* II: *Eigensystems*, SIAM, Philadelphia, 2001.
[26] R. C. Whaley and A. Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience, 35 (2005), pp. 101–121, also online at http://www.cs.utsa.edu/˜whaley/papers/spercw04.ps.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.