Generating extreme-scale matrices with specified singular values or condition numbers

Fasi, Massimiliano and Higham, Nicholas J.

2020

MIMS EPrint: **2020.8**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

GENERATING EXTREME-SCALE MATRICES WITH SPECIFIED SINGULAR VALUES OR CONDITION NUMBERS*

MASSIMILIANO FASI† AND NICHOLAS J. HIGHAM‡

Abstract. A widely used form of test matrix is the randsvd matrix constructed as the product $A = U\Sigma V^*$, where U and V are random orthogonal or unitary matrices from the Haar distribution and Σ is a diagonal matrix of singular values. Such matrices are random but have a specified singular value distribution. The cost of forming an $m \times n$ randsvd matrix is $m^3 + n^3$ flops, which is prohibitively expensive at extreme scale; moreover, the randsvd construction requires a significant amount of communication, making it unsuitable for distributed memory environments. By dropping the requirement that U and V be Haar distributed and that both be random, we derive new algorithms for forming A that have cost linear in the number of matrix elements and require a low amount of communication and synchronization. We specialize these algorithms to generating matrices with specified 2-norm condition number. Numerical experiments show that the algorithms have excellent efficiency and scalability.

Key words. test matrix, random matrix, randsvd, singular value decomposition, 2-norm condition number, Householder reflector

AMS subject classifications. 65F35, 65Y05.

1. Introduction.

The TOP500 list[1] ranks the world's fastest computers using the High-Performance Linpack Benchmark (HPL) [24], which measures the time taken to solve a dense linear system with random coefficients by means of a direct factorization method. The more powerful the machine being tested, the larger the test matrix must be in order to extract the best performance from the system.

The largest linear systems solved on Fugaku, the machine that heads the June 2020 TOP500 list, have order 10^7, and the systems needed to benchmark the coming generation of exascale supercomputers will be even larger. HPL employs random matrices with entries drawn from the uniform distribution on $(-1/2, 1/2]$. For such matrices, the expected value of the logarithm of $\kappa_2(A)$ is bounded by $4 \log n + 1$ [8, sect. 3.2], so these matrices are potentially ill conditioned for today's most extreme cases.

In the future, in HPL and other benchmarks we may wish to generate random matrices with a specified singular value distribution or a specified condition number. For example, solving a linear system by low precision LU factorization with iterative refinement at higher precisions can be faster than solving entirely at double precision [6], [7], [12], but for convergence to be guaranteed there may be a limit $\kappa_2(A) \leq \kappa_{\max}$, where $\kappa_{\max} \ll u^{-1}$ and u is the unit roundoff of the working precision ($u^{-1} \approx 10^{16}$ for IEEE double precision arithmetic [19]). The new HPL-AI mixed-precision benchmark [11] uses this approach, and in such a case we need to generate matrices with $\kappa_2(A)$ suitably bounded.

For large dimensions, existing techniques for generating dense matrices with a

*Version of October 19, 2020.

Funding: This work was supported Engineering and Physical Sciences Research Council grant EP/P020720/1 and the Royal Society.

†Department of Mathematics, The University of Manchester, Manchester, M13 9PL, UK (massimiliano.fasi@manchester.ac.uk). Present address: School of Science and Technology, Örebro University, Örebro, 701 82, Sweden (massimiliano.fasi@oru.se).

‡Department of Mathematics, The University of Manchester, Manchester, M13 9PL, UK (nick.higham@manchester.ac.uk).

[1] http://www.top500.org

chosen singular value distribution are too expensive, as their cost significantly exceeds the cost of solving $Ax = b$. A new approach is therefore needed. In order to be efficient when run at extreme scale on machines with a high degree of parallelism, the algorithms we develop must be concurrent and require a low amount of communication and synchronization. As our primary goal is to provide test cases, it is also desirable that generating the matrix takes only a fraction of the execution time of the method being tested. This is the first in the following list of requirements.

- R1. The cost of generating the matrix, measured by the number of floating-point operations (flops) required, should be linear in the number of entries.
- R2. The matrix should not be a low rank perturbation of a diagonal matrix (such a matrix is too special).
- R3. The matrix should be well scaled in the sense that there do not exist diagonal matrices D_1 and D_2 such that $\kappa_2(D_1AD_2) \ll \kappa_2(A)$. (Such diagonal scalings lead to artificial ill conditioning that some algorithms automatically remove or are insensitive to.)
- R4. The matrix should not have a large growth factor for LU factorization with pivoting (otherwise the factorization will suffer from numerical instability, which will affect the interpretation of results for such solution methods).
- R5. $\kappa_\infty(A)$ should not be especially far from $\kappa_2(A)$. We know that $n^{-1}\kappa_2(A) \leq \kappa_\infty(A) \leq n\kappa_2(A)$ [17, sect. 6.2], and since $\kappa_\infty(A)$ rather than $\kappa_2(A)$ is often used in error bounds it is preferable that $\kappa_\infty(A)$ and $\kappa_2(A)$ differ by substantially less than a factor n.

In this work we design an algorithm that constructs random dense matrices with specified singular values and is suitable for a distributed memory environment. The algorithm comes in two different forms, the most efficient of which depends on the shape of the matrix. We also derive a cheaper variant that, given the required 2-norm condition number, chooses the singular values in a way that minimizes the overall computational cost. Our algorithms satisfy requirements R1 and R2, and we will show experimentally that they satisfy R3–R5 for suitable choices of the singular value distribution.

In the next section we describe the method used by several software packages to produce random matrices with specified singular values. In section 3 we show how the cost of this technique can be substantially reduced so as to obtain practical algorithms for extreme-scale architectures. In section 4 we show that further cost reductions are possible if one is interested only in the 2-norm condition number of the matrix that is generated, and not in the distribution of its singular values. Finally, we compare these algorithms experimentally in section 5.

We recall that a Householder matrix is defined in terms of a unit 2-norm vector u by

$$H(u) = I - 2uu^*, \qquad \|u\|_2 = 1. \tag{1.1}$$

We denote by sign the function of a real argument x for which $\operatorname{sign}(x)$ is -1 if $x < 0$ and 1 if $x \geq 0$

2. The randomized SVD construction. In order to construct an $m \times n$ matrix with given singular values $\sigma_1, \ldots, \sigma_{\min(m,n)}$ we can construct the singular value decomposition (SVD)

$$A = U\Sigma V^*, \tag{2.1}$$

Algorithm 2.1: Random matrix with specified singular values.

function $A \leftarrow \text{RANDSVD}(m, n, \sigma \in \mathbb{R}^{\min(m,n)}, \texttt{realout})$

Generate $m \times n$ real (if `realout` *is* **true**) *or complex (if* `realout` *is* **false**) *matrix A with specified singular values.*

  $\Sigma \leftarrow \text{diag}(\sigma) \in \mathbb{R}^{m \times n}$
  $A \leftarrow \text{UMULT}(\Sigma, \texttt{realout})$
  $A \leftarrow \text{UMULT}(A^*, \texttt{realout})$

function $B \leftarrow \text{UMULT}(A \in \mathbb{C}^{m \times n}, \texttt{realtransf})$

Compute action $R = QA$ of a random orthogonal (if `realtransf` *is* **true**) *or unitary (if* `realtransf` *is* **false**) *matrix Q from the Haar distribution.*

  $B \leftarrow A$
  for $k \leftarrow 2$ **to** m **do**
    $v \leftarrow \text{RANDN}(k, 1)$
    if `realtransf` **then**
      $d_{m-k+1} \leftarrow -\text{sign}(v_1)$
    else
      $v \leftarrow v + \mathrm{i} \cdot \text{RANDN}(k, 1)$
      $\theta \leftarrow \text{Arg}(v_1)$
      $d_{m-k+1} \leftarrow -\mathrm{e}^{\mathrm{i}\theta}$
    $u \leftarrow v - d_{m-k+1}\|v\|_2 e_1$
    $u \leftarrow u/\|u\|_2$
    $x \leftarrow \begin{bmatrix} 0_{m-k} \\ u \end{bmatrix}$
    $B \leftarrow B - (2x)(x^T B)$ exploiting the nonzero pattern of x.
  $B \leftarrow \text{diag}(d_1, \ldots, d_{m-1}, \text{sign}(\text{RANDN}(1, 1))) \cdot B$

where $U \in \mathbb{C}^{m \times m}$ and $V \in \mathbb{C}^{n \times n}$ are unitary and $\Sigma \in \mathbb{C}^{m \times n}$ is the diagonal matrix $\text{diag}(\sigma_1, \ldots, \sigma_{\min(m,n)})$. Stewart [26] considered real matrices and proposed taking U and V as random matrices from the Haar distribution, which is a natural uniform probability distribution on the orthogonal matrices. Let $\mathcal{N}(0, 1)$ denote the normal distribution with mean 0 and variance 1, and let the entries of $B \in \mathbb{R}^{n \times n}$ be sampled from $\mathcal{N}(0, 1)$. If the QR factorization $B = QR$ is normalized so that the diagonal elements of R are nonnegative, then Q is from the Haar distribution [4], [26]. Stewart showed that Q can be generated in factored form as $Q = DP_1P_2 \ldots P_{n-1}$, where $P_i = \text{diag}(I_{i-1}, \widehat{P}_i)$ with $\widehat{P}_i$ the Householder transformation that reduces $x_i \in \mathbb{R}^{n-i+1}$ with elements from $\mathcal{N}(0, 1)$ to $r_{ii}e_1$, $D = \text{diag}(\text{sign}(r_{ii}))$, and $r_{nn} \in \mathcal{N}(0, 1)$. Forming A by generating and applying U and V in product form requires around half as many flops as explicitly computing U and V via QR factorization and then forming the product in (2.1).

Demmel and McKenney [9] implemented Stewart's method for constructing (2.1) in LAPACK's test matrix generation suite. Higham implemented the method in MATLAB as function `randsvd` in the Test Matrix Toolbox [15], [16], and this function was subsequently incorporated into the MATLAB `gallery('randsvd',...)` function. Stewart's method is also available in Julia through the `randsvd` function in Zhang and Higham's Matrix Depot package [30].

Mezzadri [22] shows that it is straightforward to extend Stewart's construction of real random orthogonal matrices to complex unitary matrices. Algorithm 2.1 summarizes how to generate both real and complex random matrices with specified singular values and orthogonal or unitary matrices of singular vectors from the Haar distribution. In the algorithm, the function $\textsc{randn}(m, n)$ generates an $m \times n$ matrix with entries sampled from $\mathcal{N}(0, 1)$, and $\operatorname{Arg}\colon \mathbb{C} \to (-\pi, \pi]$ denotes the principal value of the argument function.

Algorithm 2.1 is not appropriate for generating large-scale matrices as it requires m^3+n^3 flops [17, sect. 28.3] and so it is unduly expensive for large m and n. Moreover, it is not well suited to a distributed-memory environment because of the high volume of data communication required by the $m-1$ matrix-vector multiplications that are performed on line 18. This communication bottleneck can be reduced, to some extent, by blocking the $m-1$ Householder vectors and applying them together. This is the approach followed by the `magma_generate_svd` function of MAGMA [10].

3. The randomized SVD construction at scale. The SVD construction in the previous section can be modified so as to reduce the cost of generating real and complex matrices with specified singular values from m^3+n^3 flops to only $\mathcal{O}(mn)$ flops. For practical purposes, it is not necessary that the orthogonal or unitary matrices of left and right singular vectors be Haar distributed, and it will usually not be necessary that both matrices are random. Therefore we will turn our attention to families of orthogonal or unitary matrices that are cheaper to construct and apply.

The simplest approach is as follows. For $m \geq n$, we can generate a matrix $A \in \mathbb{C}^{m\times n}$ by taking $U \in \mathbb{C}^{m\times n}$ with orthonormal columns and rescaling its columns by the desired singular values. The resulting matrix $A = U\Sigma$, however, has a special structure, and would be, for instance, a bad test problem for least squares solvers: the normal equations matrix $A^TA = \Sigma^2$ is diagonal, and the triangular factor of the QR factorization of A is, by construction, the diagonal matrix $D\Sigma$, where $D = \operatorname{diag}(\pm 1)$. Another reason for concern in the square case is that matrices generated in this way are not ill-conditioned in the sense of the Skeel condition number $\operatorname{cond}(A) = \| \, |A^{-1}||A| \, \|_\infty$ [17, sect. 7.2], [25], since $\operatorname{cond}(A^T) = \operatorname{cond}(\Sigma U^T) = \operatorname{cond}(U^T)$. Hence these matrices do not satisfy the requirement R3 in section 1.

We conclude that in order to obtain test matrices with desirable properties, it is necessary to use nontrivial transformations on both sides of Σ in (2.1). In order to keep the computational cost under control, we will employ a class of rectangular matrices with orthonormal columns that generalize Householder transformations.

Take $m > n$, let $\alpha \in \mathbb{C}$, $u \in \mathbb{C}^n$, $v \in \mathbb{C}^{m-n}$, and $w \in \mathbb{C}^n$, and consider the matrix

$$W := \begin{bmatrix} I_n \\ 0 \end{bmatrix} + \alpha \begin{bmatrix} u \\ v \end{bmatrix} w^* = \begin{bmatrix} I_n + \alpha u w^* \\ \alpha v w^* \end{bmatrix} \in \mathbb{C}^{m\times n}.$$

We have

$$\begin{aligned} W^*W &= (I_n + \overline{\alpha} w u^*)(I_n + \alpha u w^*) + |\alpha|^2 (v^*v) w w^* \\ &= I_n + \overline{\alpha} w u^* + \alpha u w^* + |\alpha|^2 (u^*u + v^*v) w w^*. \end{aligned} \tag{3.1}$$

If we assume that $w = u \neq 0$, we can further simplify the expression in (3.1) and obtain that W has orthonormal columns if and only if

$$2C \operatorname{Re}\alpha + |\alpha|^2 = 0, \quad \text{where } C = \frac{1}{\|u\|_2^2 + \|v\|_2^2}. \tag{3.2}$$

Algorithm 3.1: Matrix with specified singular values.

function $A \leftarrow \text{RSVDFWD}(m, n, \sigma \in \mathbb{R}^{\min(m,n)}, \texttt{realout})$

Generate $m \times n$ real (if `realout` *is* **true**) *or complex (if* `realout` *is* **false**) *matrix A with specified singular values using (3.5) or (3.6).*

$p \leftarrow \min(m, n)$

$\widetilde{Q} \leftarrow \text{ORTHOG}(m, p, \texttt{realout}) \cdot \text{diag}(\sigma)$

if `realout` **then**

$\mathbb{F} \leftarrow \mathbb{R}$

$\alpha \leftarrow -2$

else

$\mathbb{F} \leftarrow \mathbb{C}$

Choose $\theta \in \mathbb{R} \setminus \{(2k+1)\pi : k \in \mathbb{Z}\}$.

$\alpha \leftarrow -(\mathrm{e}^{\mathrm{i}\theta} + 1)$

Generate $u \in \mathbb{F}^p \setminus \{0\}$.

Generate $v \in \mathbb{F}^{n-p}$.

$\zeta \leftarrow \|u\|_2^2 + \|v\|_2^2$

$\alpha \leftarrow \alpha/\zeta$

$y \leftarrow \widetilde{Q}u$

$A \leftarrow \left[\widetilde{Q} + \overline{\alpha} y u^*, \quad \overline{\alpha} y v^*\right]$

By writing $\alpha = \rho \mathrm{e}^{\mathrm{i}\varphi}$, the condition in (3.2) can be rewritten as $\rho = -2C\cos\varphi$, which gives

$$\alpha = -C(2\cos^2\varphi + 2\mathrm{i}\sin\varphi\cos\varphi) = -C - C(\cos 2\varphi + \mathrm{i}\sin 2\varphi) = -C(1 + \mathrm{e}^{\mathrm{i}2\varphi}).$$

In other words, we can choose for α any complex number on the disc of radius C centered at $-C$. Therefore, for any choice of vectors $u \in \mathbb{C}^n \setminus \{0\}$ and $v \in \mathbb{C}^{m-n}$, the matrix

$$W(\theta, u, v) := \begin{bmatrix} I_n + \alpha u u^* \\ \alpha v u^* \end{bmatrix}, \qquad \alpha = -\frac{\mathrm{e}^{\mathrm{i}\theta} + 1}{\|u\|_2^2 + \|v\|_2^2}, \qquad \theta \in \mathbb{R}, \tag{3.3}$$

has orthonormal columns. In order to avoid the trivial case $\alpha = 0$, we restrict θ to the set $\mathbb{R} \setminus \{(2k+1)\pi : k \in \mathbb{Z}\}$.

A similar argument for the real case shows that for any $u \in \mathbb{R}^n \setminus \{0\}$ and $v \in \mathbb{R}^{m-n}$ the matrix

$$Z(\alpha, u, v) := \begin{bmatrix} I_n + \alpha u u^T \\ \alpha v u^T \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{3.4}$$

has orthonormal columns if and only if $\alpha = -2C$ or $\alpha = 0$, where C is defined in (3.2).

These results can be readily used to develop an algorithm for constructing an $m \times n$ random matrix with specified singular values. First we consider the case of matrices with complex entries.

Let $p = \min(m, n)$, let $Q \in \mathbb{C}^{m \times p}$ have orthonormal columns (we will discuss how to choose Q at the end of the section), let $\Sigma \in \mathbb{R}^{p \times p}$ be a diagonal matrix of singular values, and let $W := W(\theta, u, v) \in \mathbb{C}^{n \times p}$ for some $u \in \mathbb{C}^p \setminus \{0\}$ and $v \in \mathbb{C}^{n-p}$. We

stress that the construction we present here requires only that u have at least one nonzero element, and that no assumptions on v are necessary. In the experiments in section 5 we will draw the entries of these two vectors from a Gaussian distribution, but this is not necessary, and one might want to choose u and v in a non-random way when reproducibility is required. For this reason, in our pseudocode we simply state that the two vectors have to be generated and we do not specify how their elements should be chosen.

The matrix

$$A := Q\Sigma W^* = \left[Q\Sigma + \overline{\alpha}Q\Sigma uu^*, \quad \overline{\alpha}Q\Sigma uv^*\right] \in \mathbb{C}^{m\times n}, \quad \alpha = -\frac{\mathrm{e}^{\mathrm{i}\theta}+1}{\|u\|_2^2+\|v\|_2^2}, \tag{3.5}$$

has the diagonal entries of Σ as singular values.

For the real case, the matrix

$$A := Q\Sigma Z^T = \left[Q\Sigma + \alpha Q\Sigma uu^T, \quad \alpha Q\Sigma uv^T\right], \qquad \alpha = -\frac{2}{\|u\|_2^2+\|v\|_2^2}, \tag{3.6}$$

where $Q \in \mathbb{R}^{m\times n}$ has orthonormal columns and $Z := Z(\alpha, u, v) \in \mathbb{R}^{n\times p}$ for some $u \in \mathbb{R}^p \setminus \{0\}$ and $v \in \mathbb{C}^{n-p}$, has the diagonal entries of Σ as singular values.

The matrices in (3.5) and (3.6) can be constructed efficiently as shown in Algorithm 3.1. As this strategy evaluates the two matrix products in (3.5) and (3.6) from left to right, i.e., in a forward fashion, we use the expression "forward algorithm" to refer to it. In the pseudocode, calling ORTHOG$(m, p, \texttt{realout})$ returns, for $m > p$, an $m \times p$ matrix with orthonormal columns whose elements are either real or complex depending on the value of the third input argument. If $m = n$, so that $n - p = 0$, then one can either assume, for consistency, that the vector v is 0×1 or replace lines 12–16 with

12 $\alpha \leftarrow \alpha/\|u\|_2$
13 $A \leftarrow \widetilde{Q} + (\widetilde{Q}u)(\overline{\alpha}u^*)$

where the parentheses are used to indicate the evaluation order that yields the smallest flop count. The best strategy to evaluate the expression on line 16 depends on the values of m and n: if $m < n$, then it is convenient to compute $\overline{\alpha}y$, store it, and use it to evaluate both blocks of A, whereas if $n < m$ the optimal flop count is obtained by computing $\overline{\alpha}u^*$ and $\overline{\alpha}v^*$ and using those to compute the outer products.

Let $\gamma(m, n) \in \mathcal{O}(mn)$ be the number of flops required by ORTHOG to generate an $m \times n$ matrix with orthonormal columns. Then generating the matrix and applying the diagonal scaling on line 3 requires $\gamma(m, p) + mp$ flops; the cost of the **if** statement starting on line 4 and the division on line 14 do not depend on m or n; generating the vectors on lines 11 and 12, which we assume to be random, has cost ψn, where ψ is the number of flops required to generate a single random number; and computing the norms on line 13 costs $2n - 1$ flops. The matrix-vector product on line 15 and the outer products on line 16 require $2mp - m$ and $mn + mp + p$ flops, respectively. Therefore, the total flop count of Algorithm 3.1 is

$$\begin{aligned} C_f(m, n) &= mn + \gamma(m, p) + 4mp - m + (\psi + 2)n + p + \mathcal{O}(1) \\ &\approx mn + \gamma(m, p) + 4mp. \end{aligned} \tag{3.7}$$

The only operations in Algorithm 3.1 that require communication are the matrix-vector product on line 15 and the outer products on line 16. In a distributed-memory implementation, this algorithm requires synchronization of the processes only between

Algorithm 3.2: Matrix with specified singular values.

1 **function** $A \leftarrow \text{RSVDBWD}(m, n, \sigma \in \mathbb{R}^{\min(m,n)}, \texttt{realout})$

Generate $m \times n$ real (if `realout` *is* **true**) *or complex (if* `realout` *is* **false**) *matrix A with specified singular values using (3.8) or (3.9).*

2 $p \leftarrow \min(m, n)$

3 $\widetilde{Q} \leftarrow \text{ORTHOG}(n, p, \texttt{realout}) \cdot \text{diag}(\sigma)$

4 **if** `realout` **then**

5 $\mathbb{F} \leftarrow \mathbb{R}$

6 $\alpha \leftarrow -2$

7 **else**

8 $\mathbb{F} \leftarrow \mathbb{C}$

9 Choose $\theta \in \mathbb{R} \setminus \{(2k+1)\pi : k \in \mathbb{Z}\}$.

10 $\alpha \leftarrow -(\mathrm{e}^{\mathrm{i}\theta} + 1)$

11 Generate $u \in \mathbb{F}^p \setminus \{0\}$.

12 Generate $v \in \mathbb{F}^{m-p}$.

13 $\zeta \leftarrow \|u\|_2^2 + \|v\|_2^2$

14 $\alpha \leftarrow \alpha/\zeta$

15 $y \leftarrow \widetilde{Q}u$

16 $A \leftarrow \begin{bmatrix} \widetilde{Q}^* + \alpha u y^* \\ \alpha v y^* \end{bmatrix}$

these two lines, as neither of the outer products on line 16 can be evaluated before the computation of y has been completed. This represents, however, a great improvement over Algorithm 2.1, where the processes need to synchronize after each of the $m-1$ matrix-vector products on line 18.

Alternatively one could consider, instead of (3.5), the SVD

$$A := W\Sigma Q^* = \begin{bmatrix} \Sigma Q^* + \alpha u u^* \Sigma Q^* \\ \alpha v u^* \Sigma Q^* \end{bmatrix}, \quad \alpha = -\frac{\mathrm{e}^{\mathrm{i}\theta} + 1}{\|u\|_2^2 + \|v\|_2^2}, \tag{3.8}$$

where $W := W(\theta, u, v) \in \mathbb{C}^{m \times p}$ for some $u \in \mathbb{C}^p \setminus \{0\}$ and $v \in \mathbb{C}^{m-p}$, $\Sigma \in \mathbb{R}^{p \times p}$ is a diagonal matrix of singular values, and $Q \in \mathbb{C}^{n \times p}$ has orthonormal columns. If a matrix with real entries is sought, one can instead use the decomposition

$$A := Z\Sigma Q^T = \begin{bmatrix} \Sigma Q^T + \alpha u u^T \Sigma Q^T \\ \alpha v u^T \Sigma Q^T \end{bmatrix}, \qquad \alpha = -\frac{2}{\|u\|_2^2 + \|v\|_2^2}, \tag{3.9}$$

where $Z := Z(\alpha, u, v) \in \mathbb{R}^{m \times p}$ for some $u \in \mathbb{R}^p \setminus \{0\}$ and $v \in \mathbb{R}^{m-p}$, $\Sigma \in \mathbb{R}^{p \times p}$, and $Q \in \mathbb{R}^{n \times p}$ has orthonormal columns.

The matrices in (3.8) and (3.9) can be evaluated efficiently in a backward fashion, as shown in Algorithm 3.2. If $p = m$, then we can either assume, as before, that v is a 0×1 matrix, or replace lines 12–16 with

12 $\alpha \leftarrow \alpha/\|u\|_2$

13 $A \leftarrow \widetilde{Q} + \overline{\alpha} u (u^* \widetilde{Q})$

By choosing the best evaluation order for the expression on line 16, we can show

that Algorithm 3.2 has a flop count of

$$C_b(m,n) = mn + \gamma(n,p) + 4np + (\psi+2)m - n + p + \mathcal{O}(1) \approx mn + \gamma(n,p) + 4np. \tag{3.10}$$

Algorithms 3.1 and 3.2 have the same cost if they are used to construct a square matrix with specified singular values. In order to minimize the overall computational cost we should use Algorithm 3.2 when $m > n$ and Algorithm 3.1 otherwise. In terms of communication and synchronization, the requirements of Algorithm 3.2 are the same as those of Algorithm 3.1.

The choice of the matrix generated by the function ORTHOG used in Algorithms 3.1 and 3.2 makes a difference to performance and to the structure of the matrix the algorithms produce. One possibility is to use a matrix of the form (3.3) or (3.4), which would have to be explicitly formed, at the cost of mn flops. This approach has two major drawbacks if $m = n$. On the one hand, the matrices generated by the algorithms in this section would be just a rank-3 update of the diagonal matrix Σ, violating requirement R2 in section 1. On the other hand, in a distributed-memory environment the resulting algorithm would be less efficient than computing locally the entries of the matrix because of the high communication cost associated with generating, normalizing, and distributing the vectors u and v. We support this claim by means of numerical experiments in section 5.2.1.

We will choose Q to be a matrix whose elements are given by explicit formulae that can be evaluated independently, without communication. Various such unitary and orthogonal matrices exist in the literature, including the Fourier matrix ("the most important of all unitary matrices" [27]); the Helmert matrix [13] and its generalizations [20], [21]; the Hartley matrix [3]; and matrices associated with the discrete sine [3] and the cosine [28] transform. Some orthogonal structured matrices are also discussed in the literature: most of the matrices above, as well as [18, eqs. (2.3) and (2.4)] and [2, eq. (3.2)] are symmetric orthogonal.

4. Large matrices with specified 2-norm condition number. Algorithms 3.1 and 3.2 can readily be used to construct a dense square matrix $A \in \mathbb{C}^{n\times n}$ with specified 2-norm condition number κ by choosing $\sigma \in \mathbb{R}^n$ so that $\sigma_1\sigma_n^{-1} = \kappa$. We now show how a suitable choice of Σ and u in (3.5) and (3.8) can lead to algorithms with a lower computational cost.

Since $m = n$, (3.5) simplifies to $A = Q\Sigma(I + \overline{\alpha}uu^*)$. Denoting the ith row of $Q \in \mathbb{C}^{n\times n}$ by $\mathbf{q}_{(i)}^T$, where $\mathbf{q}_{(i)} \in \mathbb{C}^n$, we can write the ith entry of the vector $y := Q\Sigma u$ computed on line 15 of Algorithm 3.1 as

$$y_i = (Q\Sigma u)_i = \mathbf{q}_{(i)}^T \Sigma u = \sum_{k=1}^{n} q_{ik}\sigma_k u_k.$$

Since Q is unitary, by setting $u = \overline{\mathbf{q}_{(\ell)}}$, for some ℓ between 1 and n, we obtain

$$y_i = \sum_{k=1}^{n} q_{ik}\sigma_k\overline{q_{\ell k}} = \sum_{k=1}^{n} q_{ik}\sigma_k\overline{q_{\ell k}} + \left(\delta_{i\ell} - \sum_{k=1}^{n} q_{ik}\overline{q_{\ell k}}\right) = \sum_{k=1}^{n} q_{ik}(\sigma_k - 1)\overline{q_{\ell k}} + \delta_{i\ell}, \tag{4.1}$$

where $\delta_{i\ell}$ denotes the Kronecker delta, which equals 1 if $i = \ell$ and 0 otherwise.

Equation (4.1) shows that the number of flops required to compute y can be reduced by setting as many entries of σ as possible to 1. Since in our application we

Algorithm 4.1: Matrix with specified 2-norm condition number.

1 **function** $A \leftarrow$ SVDCONDFWD$(n, \kappa,$ `realout`$)$

Generate $n \times n$ real (if `realout` *is* **true**$)$ *or complex (if* `realout` *is* **false**$)$ *matrix A with specified 2-norm condition number κ.*

2 $\widetilde{Q} \leftarrow$ ORTHOG$(n, n,$ `realout`$)$
3 **if** `realout` **then**
4 $\alpha \leftarrow -2$
5 **else**
6 Choose $\theta \in \mathbb{R} \setminus \{(2k+1)\pi : k \in \mathbb{Z}\}$.
7 $\alpha \leftarrow -(\mathrm{e}^{\mathrm{i}\theta} + 1)$
8 Choose ℓ between 1 and n.
9 Choose σ_1 and σ_n using one of the three strategies in (4.2).
10 $\xi_1 \leftarrow \overline{q_{\ell 1}}(\sigma_1 - 1)$
11 $\xi_n \leftarrow \overline{q_{\ell n}}(\sigma_n - 1)$
12 **for** $i \leftarrow 1$ **to** n **do**
13 $y_i \leftarrow q_{i1}\xi_1 + q_{in}\xi_n + \delta_{i\ell}$
14 **for** $i \leftarrow 1$ **to** n **do**
15 $q_{i1} \leftarrow \sigma_1 q_{i1}$
16 $q_{in} \leftarrow \sigma_n q_{in}$
17 $A \leftarrow Q + \overline{\alpha} y \mathbf{q}_{(\ell)}^T$

are interested only in the ratio of the largest to the smallest singular values, we will set $\sigma_k = 1$ for k between 2 and $n-1$. Then (4.1) becomes

$$y_i = (\sigma_1 - 1)q_{i1}\overline{q_{\ell 1}} + (\sigma_n - 1)q_{in}\overline{q_{\ell n}} + \delta_{i\ell}.$$

There are three natural choices of the extremal singular values:

$$\text{(4.2)} \qquad \sigma_1 = \kappa^{1/2}, \quad \sigma_n = \kappa^{-1/2}; \qquad \sigma_1 = \kappa, \quad \sigma_n = 1; \qquad \sigma_1 = 1, \quad \sigma_n = \kappa^{-1}.$$

After scaling $A \leftarrow \kappa^{-1/2}A$ in the first case and $A \leftarrow \kappa^{-1}A$ in the second, the complete sets of singular values become

$$\text{(4.3)} \qquad \begin{array}{llll} \sigma_1 = 1, & \sigma_k = \kappa^{-1/2}, & k = 2, \ldots, n-1, & \sigma_n = \kappa^{-1}; \\ \sigma_1 = 1, & \sigma_k = \kappa^{-1}, & k = 2, \ldots, n-1, & \sigma_n = \kappa^{-1}; \\ \sigma_1 = 1, & \sigma_k = 1, & k = 2, \ldots, n-1, & \sigma_n = \kappa^{-1}; \end{array}$$

which for $\kappa \gg 1$ can be described as "many moderately small singular values", "one large singular value", and "one small singular value" , respectively. Which choice is best will depend on the application.

Algorithm 4.1 implements this approach. As the primary application of this algorithm will be the generation of matrices with prescribed condition number, the function `svdcond_fwd` takes as input only the order and the condition number of the matrix to be generated, and then chooses the distribution of the eigenvalues using any of the three singular values distributions in (4.3).

Generating the matrix on line 2 requires $\gamma(n)$ flops, where $\gamma(n)$ denotes the number of flops required by ORTHOG to generate an $n \times n$ orthogonal or unitary matrix. The

(a) *Blocks assigned to each process.* (b) *Data dependencies for* P_{22}.

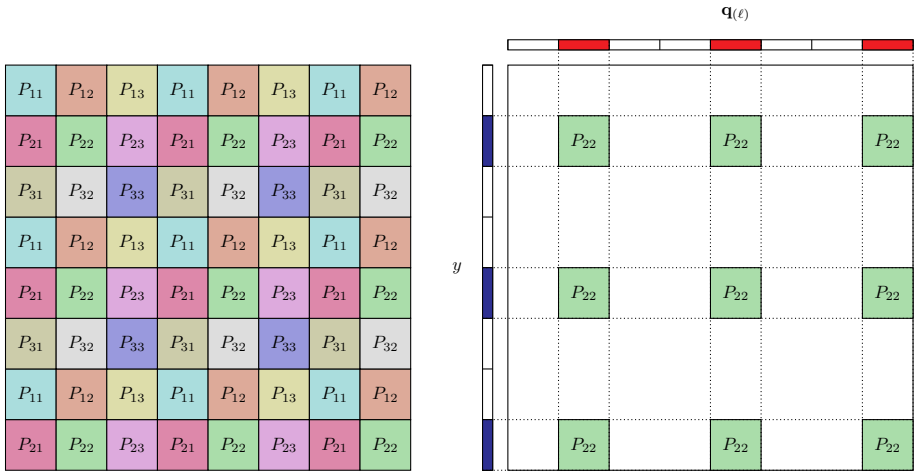

Fig. 4.1: *Left: two-dimensional block-cyclic distribution of the entries of a block* 8×8 *matrix among the processes of a* 3×3 *grid. Right: entries of the vectors* y *and* $\mathbf{q}_{(\ell)}$ *that the process* P_{22} *needs to compute in order to avoid communication when generating its portion of the test matrix using* SVDCONDFWD.

for loops on lines 12 and 14 require $4n$ and $2n$ flops, respectively, the outer product on line 17 entails n^2 additions and $(n+1)n$ multiplications, and the cost of all the other operations is constant with respect to n. Therefore, Algorithm 4.1 requires

$$\gamma(n) + 2n^2 + 8n + \mathcal{O}(1) \approx \gamma(n) + 2n^2$$

flops to generate a matrix of order n with specified 2-norm condition number. For comparison, Algorithms 3.1 and 3.2 require

$$C_f(n,n) = C_b(n,n) \approx \gamma(n) + 5n^2$$

flops to construct a matrix with the same characteristics.

Algorithm 4.1 is cheaper than Algorithm 3.1 not only in terms of computational cost, but also in terms of communication, as we now explain. Since we are interested in developing algorithms for high-performance distributed-memory environments, we consider only the case of in-core matrices, and focus on the data distribution model of BLACS/PBLAS/ScaLAPACK, as we rely on these libraries in our numerical experiments. In particular, we assume that the computational processes are arranged in a rectangular $m_p \times n_p$ grid [5, sect. 4.1] and that the entries of vectors and matrices are assigned to them according to the two-dimensional block-cyclic distribution [5, sect. 4.3.1]. Figure 4.1a shows how the elements of a block 8×8 matrix are assigned to the processes of a 3×3 grid using this scheme. The elements of column and row vectors are assigned in a similar fashion to the processes in the first column and row of the process grid, respectively.

If this model is used, the matrix-vector product on line 15 of Algorithm 3.1 requires two stages of communication. Before the computation begins, the process P_{i1}, for

$i = 1, \ldots, m_p$, distributes its entries of the vector u to the processes P_{ij}, for $j = 2, \ldots, n_p$. Then each process P_{ij} in the grid computes the product between the local matrix blocks and the corresponding blocks of u, and sends the result back to the process P_{i1}, which computes the entries of y corresponding to its own entries of u. Finally, each process in the first column distributes the entries of y to all the processes in its row, in order to compute the outer product on line 16. Overall, this algorithm requires $3m_p(n_p - 1)$ point-to-point messages for a total of $3n(n_p - 1)$ words moved.

In Algorithm 4.1, this operation is replaced by the instruction on line 13, which only requires processes in the first row of the process grid to obtain the first and last element of the ℓth column of Q and the element of the last row of Q corresponding to the column of Q that are assigned to it. The need to synchronize the processes after this computation remains, since the elements computed on line 13 may be needed by other processes in order to evaluate the outer product on line 17. Therefore, this algorithm requires $n_p + 2(n_p - 1) + n_p(m_p - 1)$ messages for a total of $3n - 2 + n(m_p - 1)$ words moved.

It is possible, however, to implement Algorithm 4.1 so that the amount of communication needed to construct the matrix depends only on the number of processes involved in the computation, but not on the order of the matrix to be generated. This comes at the price of a linear increase in the overall flop count.

In Algorithm 4.1, the operation on line 13 always requires some communication among the processes as long as the process grid has at least two rows. Similarly, computing the outer product on line 17 requires the elements of y to be communicated among the processes if the grid has two or more columns.

Let us now focus on the computation of a single entry of the matrix A generated by Algorithm 4.1. From line 17 we have that $a_{ij} = q_{ij} + \overline{\alpha} y_i q_{\ell j}$. The process in charge of the element a_{ij} computes q_{ij} on line 2 and thus can retrieve its value without any communication, but in general may not have access to y_i and $q_{\ell j}$. Even though it would be possible to obtain these values from the processes that have computed them, it may be more efficient to recompute them locally: evaluating a single element of y or $\mathbf{q}_{(\ell)}$ requires only a constant number of flops, and since the elements of the matrix are distributed by blocks, a linear amount of extra work can be used to produce a quadratic number of entries of A, as illustrated in Figure 4.1b.

Much like Algorithm 3.1, Algorithm 4.1 is a forward algorithm, as it computes the two matrix products in (3.5) and (3.6) from left to right. A backward variant of Algorithm 4.1 based on Algorithm 3.2 can be derived similarly by setting u in (3.8) and (3.9) to $\mathbf{q}^{(\ell)}$, the ℓth column of Q. The pseudocode of this approach is given in Algorithm 4.2.

We comment on requirement R3 of well scaling in section 1. This condition is difficult to check theoretically. However, we note that if $A = X\Sigma Y^T$ with X and Y orthogonal then $a_{ij} = \sum_{i=1}^{p} x_{ik}\sigma_k y_{jk}$, $p = \min(m, n)$. Hence $|a_{ij}| \leq \sigma_1$ and unless the x_{ik}, σ_k, and y_{jk} are highly correlated, we expect $|a_{ij}| \geq c_p\sigma_1$ for some constant $c_p \leq 1$ depending only on p. Hence we do not expect a wide variation in the size of the $|a_{ij}|$ and hence do not expect A to have a severe row or column scaling.

5. Numerical experiments. Now we investigate the numerical behavior of the new algorithms in sections 3 and 4 and compare their performance with that of Algorithm 2.1, which uses random orthogonal matrices from the Haar distribution.

5.1. Small-scale tests. First we consider the numerical properties of the test matrices generated by these algorithms. The small-scale experiments discussed here were run using the GNU/Linux (glnxa64) version of MATLAB 9.7.0 (R2019b Update 2)

Algorithm 4.2: Matrix with specified 2-norm condition number.

function $A \leftarrow$ SVDCONDBWD$(n, \kappa, \texttt{realout})$

Generate $n \times n$ real (if `realout` *is* **true**) *or complex (if* `realout` *is* **false**) *matrix A with specified 2-norm condition number.*

$\widetilde{Q} \leftarrow$ ORTHOG$(n, n, \texttt{realout})$
if `realout` **then**
  $\alpha \leftarrow -2$
else
  Choose $\theta \in \mathbb{R} \setminus \{(2k+1)\pi : k \in \mathbb{Z}\}$.
  $\alpha \leftarrow -(\mathrm{e}^{\mathrm{i}\theta} + 1)$
Choose ℓ between 1 and n.
Choose σ_1 and σ_n using one of the three strategies in (4.3).
$\xi_1 \leftarrow \overline{q_{1\ell}}(\sigma_1 - 1)$
$\xi_n \leftarrow \overline{q_{n\ell}}(\sigma_n - 1)$
for $i \leftarrow 1$ **to** n **do**
  $y_i \leftarrow q_{1i}\xi_1 + q_{ni}\xi_n + \delta_{i\ell}$
for $i \leftarrow 1$ **to** n **do**
  $q_{1i} \leftarrow \sigma_1 q_{1i}$
  $q_{ni} \leftarrow \sigma_n q_{ni}$
$A \leftarrow Q + \alpha \mathbf{q}^{(\ell)} y^T$

on a machine equipped with an Intel processor I5-6500 running at 3.20 GHz and 16 GiB of RAM. We test several algorithms to generate a real square matrix of order `n` with 2-norm condition number `kappa`:

- `rsvd`: a call to `gallery('randsvd',n,kappa,mode)`, the built-in MATLAB function that implements Algorithm 2.1. The distribution of the singular values depends on the parameter `mode`, which can take any of the values in Table 5.1 except 0.
- `rsvd_fwd`: an implementation of Algorithm 3.1 where $m = n$, the entries of u are sampled from $\mathcal{N}(0, 1)$, and v is a 0×1 vector. By default, the matrix Q is an orthogonal matrix from the Haar distribution. The parameter `mode` can take any of the values in Table 5.1.
- `rsvd_bwd`: an implementation of Algorithm 3.2 that uses the same parameters as `rsvd_fwd`.
- `svdcond_fwd`: an implementation of Algorithm 4.1 where the parameter ℓ is an integer drawn with uniform probability from the set $\{1, 2, \ldots, n\}$, and the matrix Q is generated as in `rsvd_fwd`. The parameter `mode` can take only the three values 0, 1, or 2, which correspond to the modes in Table 5.1 (or more precisely to the choices in (4.3), since we need $\sigma_k = 1$ for k between 2 and $n - 1$ (cf. (4.3)).
- `svdcond_bwd`: an implementation of Algorithm 4.2 that uses the same parameters as `svdcond_fwd`.

The implementation of `rsvd_fwd`, `rsvd_bwd`, `svdcond_fwd`, and `svdcond_bwd` we used for the tests is available on the MATLAB Central File Exchange.[2]

[2] https://uk.mathworks.com/matlabcentral/fileexchange/74485.

Table 5.1: *Distribution of the singular values for different values of the parameter* `mode`.

`mode`	Singular values			
0	$\sigma_1 = 1,$	$\sigma_k = \kappa^{-1/2},$	$k = 2, \dots, n-1,$	$\sigma_n = \kappa^{-1}$
1	$\sigma_1 = 1,$	$\sigma_k = \kappa^{-1},$	$k = 2, \dots, n$	
2		$\sigma_k = 1,$	$k = 1, \dots, n-1,$	$\sigma_n = \kappa^{-1}$
3		$\sigma_k = \kappa^{\frac{1-k}{n-1}},$	$k = 1, \dots, n$	
4		$\sigma_k = \frac{1}{\kappa}\big(1 + \frac{\kappa-1}{n-1}(k-1)\big),$	$k = 1, \dots, n$	
5		$\sigma_k = \exp(-\gamma_k \log \kappa),$	$k = 1, \dots, n,$	$\gamma_k \sim U[0,1]$

5.1.1. Conditioning. This experiment is designed to test how the matrices that these algorithms generate compare with respect to different measures of sensitivity. Figure 5.1 reports the value of the ratio $\mu(A) = \kappa_\infty(A)/\kappa_2(A)$ where A is a matrix of order 1,000 generated by `rsvd`, `rsvd_fwd`, `rsvd_bwd`, `svdcond_fwd`, and `svdcond_bwd` with the parameter `kappa` set to 10^i, for $i = 1, \dots, 15$. As some of the parameters used by the algorithms are chosen randomly, we take the average value of $\mu(A)$ over 20 generated matrices. The results in each of the six plots refer to a different distribution of the singular values.

The figure shows that that the ratio depends not only on the algorithm used to generate the test matrix, but also on which singular value distribution is chosen. Generally speaking, as $\kappa_2(A)$ increases the value of $\mu(A)$ follows one of three patterns: it remains constant, it has a sharp initial drop followed by a fairly constant regime, or it decreases at a sublinear rate.

The only algorithms that always exhibit the first kind of behaviour are `rsvd_bwd` and `svdcond_bwd`: both methods always generate matrices with an ∞-norm condition number larger than the 2-norm condition number in our experiments. We remark that for matrices of order 1,000 the maximum theoretical value of $\mu(A)$ is $n^2 = 10^6$, and a value between 20 and 30 can still be regarded as small.

The forward methods `rsvd_fwd` and `svdcond_fwd`, on the other hand, typically achieve the smallest values of $\mu(A)$, the only exception being when `mode` is 2. This suggests that for these two methods the conditioning in the sense of the ∞-norm is sensitive to a large gap between the two largest singular values, but not between the smallest two. We note that these are the only two of the new algorithms for which $\mu(A)$ falls below 1, which indicates matrices that are more ill conditioned in the 2-norm than in the ∞-norm sense.

Finally, for `rsvd` the value of this ratio is typically in-between those of `rsvd_fwd` and `rsvd_bwd`.

As mentioned above, in these experiments Q is a Haar distributed matrix. When considering these algorithms at scale, however, performance constraints call for the use of orthogonal matrices whose elements can be generated in constant time and without requiring any communication. In order to check whether this limitation affects the results discussed in this section, we repeated the same experiments using the symmetric orthogonal matrix $Q \in \mathbb{C}^{n\times n}$ in [18, Eq. (2.3)], defined by

$$q_{ij} = \frac{2}{\sqrt{2n+1}} \sin\left(\frac{2ij\pi}{2n+1}\right), \qquad i, j = 1, \dots, n. \tag{5.1}$$

Despite the very special structure of this matrix, we found that using it in place of the default matrix in `rsvd_fwd`, `rsvd_bwd`, `svdcond_fwd`, and `svdcond_bwd` yields results

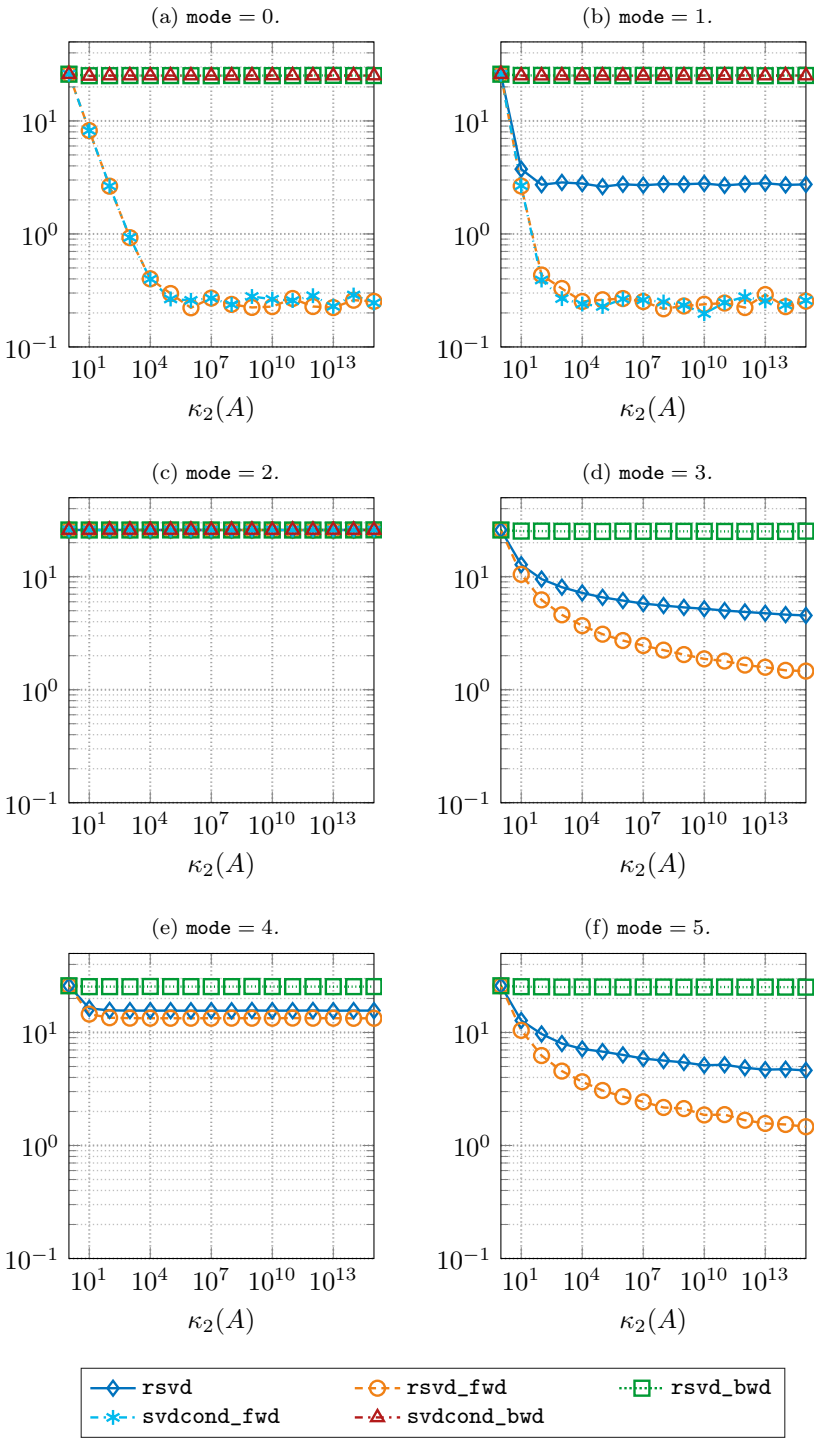

Fig. 5.1: *Ratio* $\mu(A) = \kappa_\infty(A)/\kappa_2(A)$ *of condition numbers for matrices of order* 1,000 *generated by* `rsvd`, `rsvd_fwd`, `rsvd_bwd`, `svdcond_fwd`, *and* `svdcond_bwd`.

very similar to those in Figure 5.1.

5.1.2. Growth factors. For the matrices generated we are interested in the the growth factor for LU factorization with partial pivoting, an important quantity for gauging the backward stability of the factorization as a means to solve linear systems [17, sect. 9.3], [29]. We recall that for the Gaussian elimination algorithm that transforms a matrix $A^{(0)} \in \mathbb{C}^{n\times n}$ into the upper triangular matrix $A^{(n-1)}$ by constructing the sequence of matrices $A^{(1)}$, $A^{(2)}$, ..., the growth factor is defined as

$$\rho(A) = \frac{\max_{i,j,k}|a_{ij}^{(k)}|}{\max_{i,j}|a_{ij}^{(0)}|}. \tag{5.2}$$

Seventy years of digital computing attest to the fact that the growth factor is almost always small, say less than 50.

We generated matrices of order n between 10 and 1,000, using the default choice for the orthogonal matrix Q and the six singular value distributions in Table 5.1. The growth factors were less than 50 for all choices of `mode` other than 2, and approximately 1 when mode was set to 0 or 1. When `mode` is set to 2, the growth is above 100 for all the algorithms we consider. The reasons for the large growth are investigated in [14].

Our results suggest that replacing one of the singular vector matrices in Algorithm 2.1 with the matrix in (3.3) does not influence the growth factor of the test matrices being generated.

We repeated the experiment using the matrix Q in (5.1) instead of the default choice. When `mode` is 0, 1, 2, or 3, the behavior of the four new algorithms is not influenced by the different choice of Q, although we remark that when not constant the growth factors tend to increase somewhat more quickly for an orthogonal matrix with such a special structure. The two algorithms `rsvd_bwd` and `rsvd_fwd`, on the other hand, exhibit large growth when `mode` is 4 or 5, respectively.

The conclusion to be drawn is that for LU factorization tests, `mode` should be set to 2, 4, or 5 only if large growth is wanted.

5.1.3. Scaling. It is known that the minimum value of $\kappa_2(D_1AD_2)$ over all nonsingular diagonal matrices D_1 and D_2 is bounded above by $\varrho(|A||A^{-1}|)$, where ϱ is the spectral radius, provided that $|A||A^{-1}|$ and $|A^{-1}||A|$ are irreducible [1], [17, Prob. 7.10]. We computed the ratio $\varrho\big(|A||A^{-1}|\big)/\kappa_2(A)$ for every matrix used in the experiments in section 5.1.1, and found that the behavior of this quantity varies greatly depending not only on the algorithm used to generate the matrix but also on the distribution of the singular values.

In many cases the ratio was greater than 1. If we regard a ratio less than 10^{-2} as signaling poor scaling, then for our algorithms poor scaling was observed only when `mode` was 0. Empirically, then, requirement R3 is satisfied for `rsvd_fwd`, `rsvd_bwd`, `svdcond_fwd`, and `svdcond_bwd` when `mode` is not 0.

5.1.4. Performance. Although our main focus is on distributed memory implementations, we note that the MATLAB implementations of the new algorithms are substantially faster than the existing MATLAB `gallery('randsvd',...)` function. We illustrate this with the following example. The code

```
n = 10000; kappa = 1e6; mode = 2; rng(1)
% gallery('randsvd',...)
fprintf('gallery(''randsvd'',...):   elapsed time is %5.2f seconds.\n',...
```

```
timeit(@()gallery('randsvd',n,kappa,mode,[],[],1)));
% Algorithm 3.1.
method = 1; matrix = 0;
fprintf('Alg. 3.1 with Haar Q:     elapsed time is %5.2f seconds.\n',...
        timeit(@()randsvdfast(n,kappa,mode,method,matrix)));
matrix = 2;
fprintf('Alg. 3.1 with Q in (5.1): elapsed time is %5.2f seconds.\n',...
        timeit(@()randsvdfast(n,kappa,mode,method,matrix)));
% Algorithm 4.1.
method = 3; matrix = 0;
fprintf('Alg. 4.1 with Haar Q:     elapsed time is %5.2f seconds.\n',...
        timeit(@()randsvdfast(n,kappa,mode,method,matrix)));
matrix = 2;
fprintf('Alg. 4.1 with Q in (5.1): elapsed time is %5.2f seconds.\n',...
        timeit(@()randsvdfast(n,kappa,mode,method,matrix)));
```

produces the output:

```
gallery('randsvd',...):   elapsed time is 79.52 seconds.
Alg. 3.1 with Haar Q:     elapsed time is 19.70 seconds.
Alg. 3.1 with Q in (5.1): elapsed time is  1.90 seconds.
Alg. 4.1 with Haar Q:     elapsed time is 19.28 seconds.
Alg. 4.1 with Q in (5.1): elapsed time is  1.43 seconds.
```

5.2. Large-scale tests. In this set of experiments we focus on the performance of our algorithms in a distributed-memory environment. Our codes were implemented in C using the BLACS, PBLAS, and ScaLAPACK routines provided by the Intel Math Kernel Library[3] (version 18.0.3) linked against the Open MPI[4] implementation (version 3.1.4) of version 3.1 of the MPI standard [23]. Our experiments were run on the High Performance Computing Pool of the Computational Shared Facility 3 of the University of Manchester. Each node in the pool is equipped with two 16-core Intel Xeon Gold 6130 CPUs, running at 2.10 GHz with 187 GiB of RAM, with a Mellanox Technologies ConnectX-5 InfiniBand port adaptor supporting 100Gb/s Ethernet. Some of the experiments discussed in section 5.2.3 require a single node with at least 298 GiB of RAM: for those we use a high-memory node that complements two CPUs as above with 1.5 TiB of RAM.

We compare the following codes for generating an $n \times n$ matrix with specified condition number κ.

- `prsvd`: a C port of ScaLAPACK's `pdlagge`, which is not available in the Intel MKL Library. This function returns the matrix $U\Sigma V$, where U and V are the orthogonal factors of the QR and LQ decomposition, respectively, of two matrices with random entries drawn from $(-1, 1)$. As we are interested only in the 2-norm condition number, we need to specify only the extreme singular values, and are not concerned with their distribution. Therefore, we use `mode` 3 in Table 5.1.
- `prsvd_fwd`: an implementation of Algorithm 3.1 where $m = n$, the entries of u are sampled from $\mathcal{N}(0, 1)$, v is a 0×1 vector, the matrix Q is that in (5.1) and the singular values are distributed according to `mode` 3 in Table 5.1.

[3] https://software.intel.com/mkl
[4] https://www.open-mpi.org

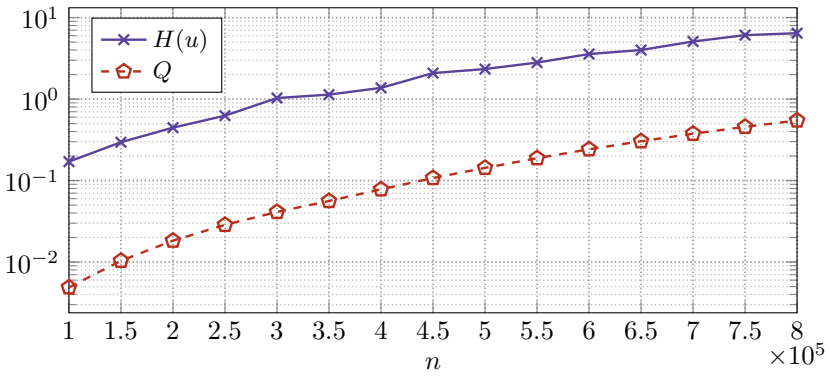

Fig. 5.2: *Wall-clock time (in seconds) required to generate the orthogonal matrices $H(u)$ and Q in (5.1) of order n between* 100,000 *and* 800,000 *using* 1,024 *processes.*

- `prsvd_bwd`: an implementation of Algorithm 3.2 using the same parameters as `prsvd_fwd`.
- `psvdcond_fwd`: a communication-avoiding implementation of Algorithm 4.1 that uses the matrix Q in (5.1). The parameter ℓ is an integer drawn with uniform probability from the set $\{1, 2, \ldots, n\}$ and `mode` is 0.
- `psvdcond_bwd`: a communication-avoiding implementation of Algorithm 4.2 that uses the same parameters as `psvdcond_fwd`.

The code we used for our experiments is available on GitHub.[5]

5.2.1. Generation of orthogonal matrices. First, we investigate the efficiency of different techniques for generating large matrices with orthonormal columns in a distributed-memory environment.

In Figure 5.2 we compare the wall-clock time necessary to generate the Householder matrix $H(u)$ in (1.1), where the parameter vector u is generated randomly, with the wall-clock time needed to construct the matrix in (5.1). We consider matrices of order between 100,000 and 800,000, and use 1,024 MPI processes.

The results clearly show the benefits of generating the matrices from a somewhat expensive formula (because of the sine evaluations) that depends only on the row and column index of the entry: this local approach is about one order of magnitude faster than the approach based on the Householder transformation, which in principle requires a much lower number of flops.

5.2.2. Scalability. In this experiment we compare the five algorithms in terms of computational efficiency. Figure 5.3 reports the wall-clock time that `prsvd`, `prsvd_fwd`, `prsvd_bwd`, `psvdcond_fwd`, and `psvdcond_bwd` require in order to generate matrices of order n between 10,000 and 100,000 using 1,024 processes. As the choice of the target condition number does not influence the run time of any of the algorithms, we set $\kappa = 10^8$ in all cases.

The results clearly show that `prsvd` is far slower than the four new methods, with a run time between one and three orders of magnitude larger than that of the second worst algorithm. Moreover, we note that the gap between this algorithm and the others grows as the order of the matrix increases. For all the sizes we consider in this

[5] https://github.com/mfasi/randsvdfast

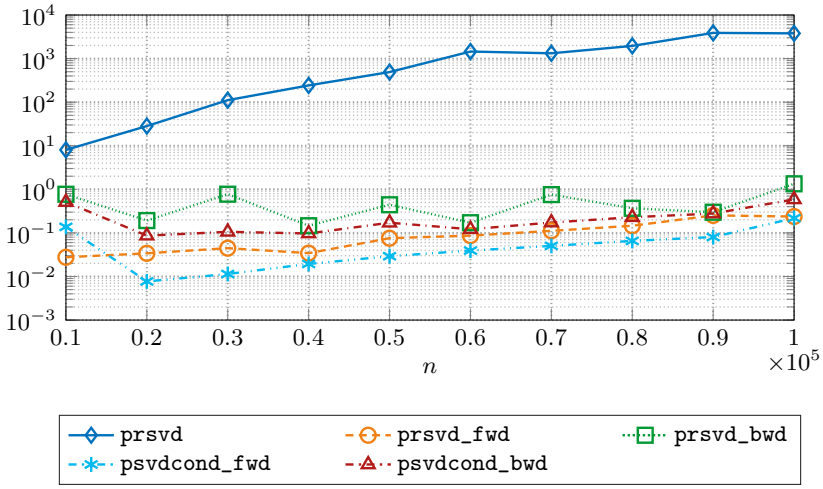

Fig. 5.3: *Wall-clock time* (*in seconds*) *required by* `prsvd`, `prsvd_fwd`, `prsvd_bwd`, `psvdcond_fwd`, *and* `psvdcond_bwd` *to generate matrices of order* n *between* 10,000 *and* 100,000 *and* 2*-norm condition number* 10^8 *using* 1,024 *processes.*

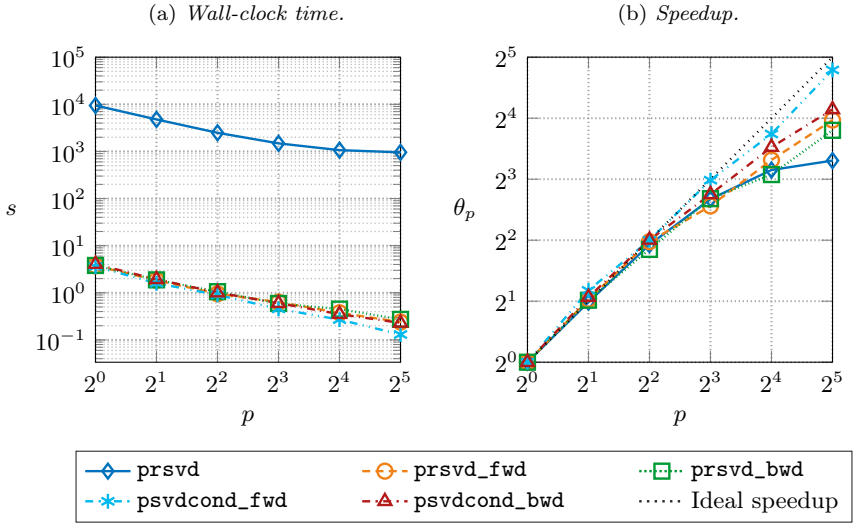

Fig. 5.4: (a) *wall-clock time* (*in seconds*) *and* (b) *speedup for* `prsvd`, `prsvd_bwd`, `prsvd_fwd`, `psvdcond_fwd`, *and* `psvdcond_bwd` *generating a matrix of order* 20,000 *and* 2*-norm condition number* 10^8 *using* p *processes.*

experiment, `psvdcond_fwd` is typically the fastest method, followed by `prsvd_fwd`, and then `psvdcond_bwd` and finally `prsvd_bwd`. As we will see in the next section, this is due to the fact that here we are dealing with somewhat small matrices, a constraint due to the exceedingly long execution time of `prsvd`. For matrices of order 200,000 or larger, the communication-avoiding algorithms `psvdcond_fwd` and `psvdcond_bwd` tend to be faster than those based on explicit matrix-matrix or matrix-vector multiplications.

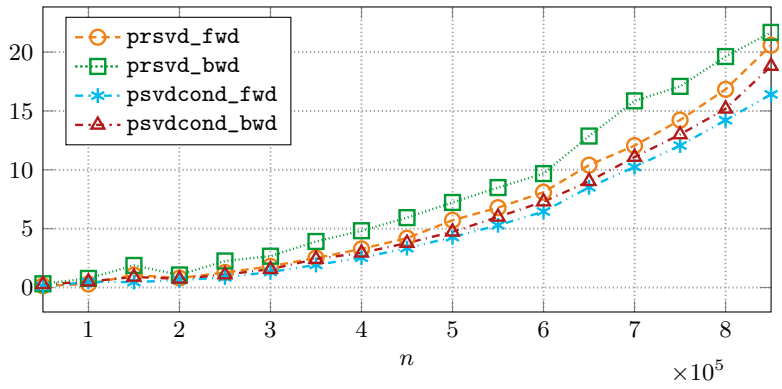

Fig. 5.5: *Wall-clock time* (*in seconds*) *required by the new algorithms* (`prsvd_bwd`, `prsvd_fwd`, `psvdcond_fwd`, *and* `psvdcond_bwd`) *to generate matrices of order* n *between* 50,000 *and* 850,000 *and* 2*-norm condition number* 10^8 *using* 1,024 *processes.*

In Figure 5.4, we investigate how well the run time of the methods scales with the number of processes being used. Figure 5.4a reports the wall-clock time needed by `prsvd`, `prsvd_fwd`, `prsvd_bwd`, `psvdcond_fwd`, and `psvdcond_bwd` to generate matrices of order 20,000 using an increasing number of processes. The use of such a small matrix in this experiment is due to the large execution time of `prsvd` with a single process.

In order to assess how increasing the number of processes involved in the computation benefits the algorithms individually, in Figure 5.4b we plot the same data as in Figure 5.4a in the form of latency speedup profiles. In other words, for each algorithm we plot, as p increases, the quantity

$$\theta_p = \frac{t_1}{t_p},$$

where t_p is the wall-clock time required to generate the matrix using p processes. For reference, we also mark the ideal speedup $\theta_p = p$, which represents the optimal theoretical performance gain of a parallel code.

It is clear from the results that the behavior of `prsvd` is qualitatively different from that of the new algorithms, which is not surprising as the scalability of this algorithm is limited by the high communication requirements that the matrix-matrix multiplication entails. Not surprisingly, the communication-avoiding algorithms outperform `prsvd_fwd` and `prsvd_bwd`, with `psvdcond_bwd` falling just a factor two short of the ideal speedup.

5.2.3. Generating large-scale matrices. For the final experiment, we test our new algorithms at scale, generating matrices of order up to 850,000, which is the order of the largest matrix we can generate with the computational resources at our disposal: such a matrix occupies about 5.26 TiB out of the 5.85 TiB of RAM available on 32 of the nodes described above.

In Figure 5.5 we report the wall-clock time required by the new algorithms to generate a matrix with 2-norm condition number $\kappa = 10^8$ and order between 50,000 and 850,000. The four algorithms appear to follow a similar trend, with `psvdcond_fwd` being always the fastest and `prsvd_bwd` always the slowest. As already mentioned, there is no clear winner among the remaining two algorithms: `prsvd_fwd` is faster

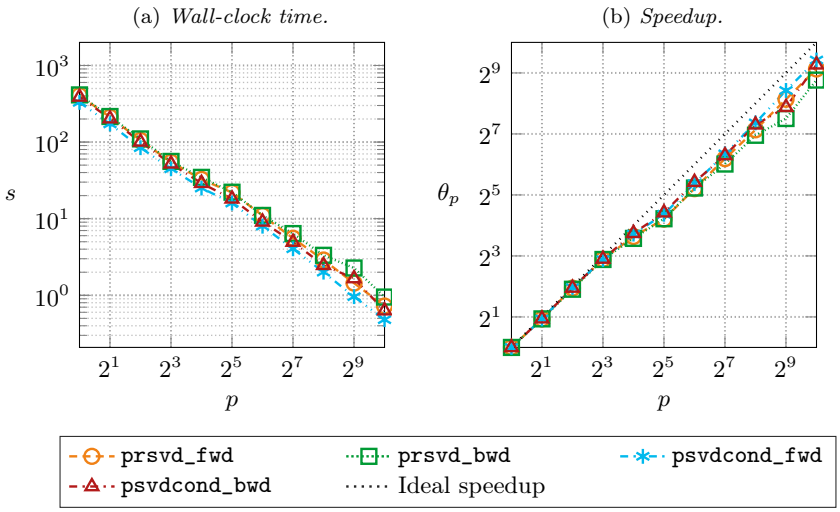

Fig. 5.6: (a) *wall-clock time* (*in seconds*) *and* (b) *speedup for the new algorithms* (`prsvd_bwd`, `prsvd_fwd`, `psvdcond_fwd`, *and* `psvdcond_bwd`) *in generating a matrix of order* 200,000 *and* 2*-norm condition number* 10^8 *using* p *processes.*

than `psvdcond_bwd` on matrices of size up to 150,000, but slower for larger matrices. This seems to suggest that the communication avoidance plays a minor role for small matrices, but becomes more and more important when larger matrices are being generated.

The scalability of the new algorithms is considered in Figure 5.6. The wall-clock time required by the four algorithms to generate a matrix of order 200,000 using 1 to 1,024 processes is shown in Figure 5.6a. The timing decreases linearly for all four algorithms; `psvdcond_fwd` is overall the fastest method and the other three are comparable but `prsvd_bwd` becomes slower than the other two when 2^7 or more processes are used. The speedup curves in Figure 5.6b show that `prsvd_fwd`, `prsvd_bwd`, `psvdcond_fwd`, and `psvdcond_bwd` all scale extremely well with the number of processes involved in the computation, as the first three fall within a factor 2 from the ideal speedup, whereas `psvdcond_bwd` is within a factor 3.

We stress that the results discussed in this section depend on the setup of the computational facility we are using, and that the picture might be different if the same experiment were repeated on a different parallel computer. In particular, it is reasonable to expect that the gap between the communication-based algorithms `prsvd_fwd` and `prsvd_bwd` and their communication-avoiding variants `psvdcond_fwd` and `psvdcond_bwd` will widen on clusters with slower network interfaces or faster CPUs.

6. Conclusions. As high performance computing moves towards exascale and beyond, generating matrices for benchmarks and testing is becoming challenging. Matrices with elements from a standard probability distribution can be efficiently formed but become more ill conditioned as the dimension grows, whereas generating matrices with specified singular values using the randsvd algorithm in Algorithm 3.1 is too expensive. We have developed modified forms of this technique that give up the property that the matrices of singular vectors are from the Haar distribution and that both are random, instead using a rectangular generalization of a Householder matrix

defined in terms of a random vector parameter together with an arbitrary orthogonal matrix, the latter intended to be one whose elements are given by an explicit formula. The algorithms require only $\mathcal{O}(mn)$ operations, as opposed to the m^3+n^3 flops for the standard randsvd approach. Two of the algorithms are specialized to the case where only the 2-norm condition number is specified and they have a reduced operation count.

Our algorithms are designed for a distributed memory environment and aim to minimize the required communication. The experiments show that they are efficient and scalable. The algorithms satisfy requirements R1 and R2 in section 1, and they satisfy R3–R5 for suitable choices of the singular value distribution.

Acknowledgments. We thank Mark Gates, Mantas Mikaitis, and Srikara Pranesh for reading early versions of this manuscript and providing helpful comments, and the two anonymous referees for their feedback that helped us improve the presentation of this manuscript. We also acknowledge the assistance given by Research IT at the University of Manchester, and the use of the HPC Pool funded by the Research Lifecycle Programme at the University of Manchester.

REFERENCES

[1] F. L. Bauer, *Optimally scaled matrices*, Numer. Math., 5 (1963), pp. 73–87.
[2] D. Bini and M. Capovani, *Spectral and computational properties of band symmetric Toeplitz matrices*, Linear Algebra Appl., 52-53 (1983), pp. 99–126.
[3] D. Bini and P. Favati, *On a matrix algebra related to the discrete Hartley transform*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 500–507.
[4] G. Birkhoff and S. Gulati, *Isotropic distributions of test matrices*, Zeitschrift für angewandte Mathematik und Physik ZAMP, 30 (1979), pp. 148–158.
[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, and et al., *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, Jan 1997.
[6] I. Buck, *World's fastest supercomputer triples its performance record.* https://blogs.nvidia.com/blog/2019/06/17/hpc-ai-performance-record-summit/, June 2019. Accessed June 24, 2019.
[7] E. Carson and N. J. Higham, *Accelerating the solution of linear systems by iterative refinement in three precisions*, SIAM J. Sci. Comput., 40 (2018), pp. A817–A847.
[8] J. A. Cuesta-Albertos and M. Wschebor, *Some remarks on the condition number of a real random square matrix*, J. Complex., 19 (2003), pp. 548–554.
[9] J. W. Demmel and A. McKenney, *A test matrix generation suite*, Preprint MCS-P69-0389, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA, Mar. 1989. LAPACK Working Note 9.
[10] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, *Accelerating numerical dense linear algebra calculations with GPUs*, in Numerical Computations with GPUs, V. Kindratenko, ed., Springer-Verlag, Cham, Switzerland, 2014, pp. 3–28.
[11] J. J. Dongarra, P. Luszczek, and Y. M. Tsai, *HPL-AI mixed-precision benchmark*. https://icl.bitbucket.io/hpl-ai/.
[12] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, *Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18 (Dallas, TX), Piscataway, NJ, USA, 2018, IEEE Press, pp. 47:1–47:11.
[13] F. R. Helmert, *Die Genauigkeit der Formel von Peters zur Berechnung des wahrscheinlichen Beobachtungsfehlers directer Beobachtungen gleicher Genauigkeit*, Astronomische Nachrichten, 88 (1876), pp. 113–131.
[14] D. J. Higham, N. J. Higham, and S. Pranesh, *Random matrices generating large growth in LU factorization with pivoting*, MIMS EPrint 2020.13, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, May 2020.

[15] N. J. Higham, *Algorithm 694: A collection of test matrices in MATLAB*, ACM Trans. Math. Software, 17 (1991), pp. 289–305.
[16] N. J. Higham, *The Test Matrix Toolbox for MATLAB (version 3.0)*, Numerical Analysis Report No. 276, Manchester Centre for Computational Mathematics, Manchester, England, Sept. 1995.
[17] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2002.
[18] N. J. Higham and D. J. Higham, *Large growth factors in Gaussian elimination with pivoting*, SIAM J. Matrix Anal. Appl., 10 (1989), pp. 155–164.
[19] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*, Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, July 2019.
[20] J. O. Irwin, *On the distribution of a weighted estimate of variance and on analysis of variance in certain cases of unequal weighting*, J. Roy. Statist. Soc., 105 (1942), p. 115.
[21] H. O. Lancaster, *The Helmert matrices*, Amer. Math. Monthly, 72 (1965), p. 4.
[22] F. Mezzadri, *How to generate random matrices from the classical compact groups*, Notices Amer. Math. Soc., 54 (2007), pp. 592–604.
[23] MPI Forum, *MPI: A Message-Passing Interface Standard, Version 3.1*, High Performance Computing Center Stuttgart (HLRS), 2015.
[24] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, *HPL: A portable implementation of the High-Performance Linpack benchmark for distributed-memory computers, Version 2.3*, 2018.
[25] R. D. Skeel, *Scaling for numerical stability in Gaussian elimination*, J. Assoc. Comput. Mach., 26 (1979), pp. 494–526.
[26] G. W. Stewart, *The efficient generation of random orthogonal matrices with an application to condition estimators*, SIAM J. Numer. Anal., 17 (1980), pp. 403–409.
[27] G. Strang, *Wavelet transforms versus Fourier transforms*, Bull. Amer. Math. Soc., 28 (1993), pp. 288–306.
[28] G. Strang, *The discrete cosine transform*, SIAM Rev., 41 (1999), pp. 135–147.
[29] J. H. Wilkinson, *Error analysis of direct methods of matrix inversion*, J. Assoc. Comput. Mach., 8 (1961), pp. 281–330.
[30] W. Zhang and N. J. Higham, *Matrix Depot: An extensible test matrix collection for Julia*, PeerJ Comput. Sci., 2 (2016), p. e58.