SANDIA REPORT
SAND2024-12753R
Printed September 2024

Bringing randomized algorithms to mainstream numerical linear algebra

Riley Murray

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from

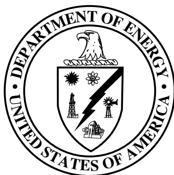
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods



ABSTRACT

Numerical linear algebra (NLA) underpins huge swaths of computational science and engineering. For scientists and engineers to make the most of the DOE's computing resources, it is essential that they have access to high-performance implementations of algorithms with best-in-class scalability and reliability. Despite this, prevailing NLA libraries have little to no support for breakthrough algorithms from the field of *randomized numerical linear algebra* (RandNLA) that have been developed over the past twenty years.

The goal of this LDRD was to break a log-jam that had prevented broad adoption of RandNLA. Our work had two thrusts. The first was to develop RandBLAS: a trustworthy and high-performance C++ library for randomized dimension reduction (an operation widely known as *sketching*). The second was the development of a novel randomized algorithm for computing a challenging type of matrix decomposition known as *Householder QR with column pivoting* (Householder QRCP).

In this one-year late-start LDRD we successfully delivered RandBLAS 1.0 and new CPU and GPU codes for Householder QRCP. RandBLAS has extensive documentation at

`https://randblas.readthedocs.io/en/stable/`.

Papers on RandBLAS and and our high-performance QRCP codes are forthcoming.

This page intentionally left blank.

CONTENTS

1. Summary **11**
- 1.1. RandBLAS 11
- 1.2. Householder QRCP 13
- 1.3. Products of this LDRD 15
- 1.4. Assessment and outlook 16

2. Discussion of technical work on RandBLAS **17**
- 2.1. Preliminaries 17
  - 2.1.1. Short and long axes of sketching operators 17
  - 2.1.2. Counter-based random number generators 18
- 2.2. Distributions, and how we sample from them 18
  - 2.2.1. Dense sketching operators 19
  - 2.2.2. Sparse sketching operators 19
- 2.3. Basic statistical tests 22
  - 2.3.1. Background: Kolmogorov-Smirnov 22
  - 2.3.2. Sampling from index sets 23
  - 2.3.3. Continuous distributions on the reals 23
- 2.4. Oblivious subspace embedding properties, and how to test for them 25
- 2.5. Sparse matrix datastructures 26
- 2.6. Multiplying sparse and dense matrices 27
  - 2.6.1. Routing for `spmm` 27
  - 2.6.2. Routing for `sketch_general` with `SparseSkOp` objects 28
  - 2.6.3. Routing for `sketch_sparse` 28

3. Discussion of technical work on QRCP **29**
- 3.1. Problem statement and background 29
- 3.2. The high-level algorithm 30
  - 3.2.1. A fast QRCP, suitable for wide sketches 31
  - 3.2.2. Column permutations 31

Appendices **35**

This page intentionally left blank.

LIST OF FIGURES

Figure 1-1. Runtime data for Algorithm 1 on sparse matrices in the SuiteSparse matrix collection, using an M2 Max MacBook Pro (12 cores, 48 MB L3 cache, 96 GB RAM) and running one OpenMP thread per core. Matrix dimensions are indicated with superimposed text. The black, blue, and red curves show data when $\ell = 0$, $\ell = 1$, and $\ell = 2$, respectively. Plots in the lower row show that algorithm runtime is dominated by BLAS and LAPACK functions rather than RandBLAS functions, and this becomes more pronounced as k increases. 13

Figure 1-2. Standardized flop rates for QR algorithms applied to large square matrices on two architectures (computed by taking the flop count of unpivoted Householder QR and dividing by algorithm runtime). The x-axis shows different choices of block size for randomized algorithms. *Left*: CPU algorithms for square matrices of order 65536. *Right*: GPU algorithms for square matrices of order 32768 (NVIDIA H100). Only two algorithms were run on the H100 because cuSOLVER does not implement a QRCP algorithm. 14

Figure 2-1. Sparsity patterns of short-axis-sparse and long-axis-sparse operators ("SASOs" and "LASOs"). Note that transposition does not affect whether an operator is a SASO or LASO. . 17

Figure 2-2. Example function for sampling k elements from the length-n array "`indices`" uniformly without replacement, using k steps of Fisher-Yates shuffling. The first for-loop performs the sampling, using a black-box function called `random_unit` that accepts integers "`lo`" and "`hi`," and a CBRNG counter and key, and returns an unsigned integer sampled uniformly at random from `lo` to `hi - 1`. The second for-loop restores `indices` to the state it had on entry. RandBLAS does not use this exact code. 20

Figure 2-3. Example function for making r independent samples of k elements from $[\![n]\!]$ uniformly without replacement. The function's total runtime is $\Theta(kr + n)$. RandBLAS does not use this exact code. 21

Figure 2-4. `vals` and `locs` are length-k buffers; the former contains independent samples from a mean-zero variance-one sub-gaussian distribution (like the Rademacher distribution) and the latter contains integers sampled uniformly with replacement from some index set. On exit, the first `num_unique_locs` entries of `locs` have been overwritten with the unique values that `locs` had on entry, and `vals[j]` is multiplied by the square root of the number of times that `locs[j]` occurred in `locs`. RandBLAS does not use this exact code. 22

Figure 2-5. Example showing how the Kolmogorov-Smirnov test statistic can be evaluated for continuous distributions in standard in C++. RandBLAS does not use this exact code. 24

Figure 2-6. Visualizations of sketch quality for a wide $d \times m$ SASO for two types of $m \times n$ data matrices, where $(d, m, n) = (6000, 100000, 2000)$. The "Gaussian" matrix has iid standard-normal entries, while the "spiked" matrix is formed by stacking copies of the identity matrix and randomly scaling n rows by 10000. Plots show mean quantities computed with 10 different SASOs for a fixed matrix. Vertical lines in the right plot show sample standard deviations. The plots have two major messages. First, some data is "easier to sketch" than others. Second, even very small values of `vec_nnz` suffice to reduce distortion to very low levels. 25

This page intentionally left blank.

LIST OF TABLES

Table 1-1. System specifications for benchmarks in the left panel of Figure 1-2. 14

This page intentionally left blank.

1. SUMMARY

Numerical linear algebra (NLA) underpins huge swaths of computational science and engineering. For scientists and engineers to make the most of the DOE's computing resources, it is essential that they have access to high-performance implementations of algorithms with best-in-class scalability and reliability. Despite this, prevailing NLA libraries have little to no support for breakthrough algorithms from the field of *randomized numerical linear algebra* (RandNLA) that have been developed over the past twenty years.

The goal of this LDRD was to break the log-jam that had prevented broad adoption of RandNLA. Our work had two thrusts. The first was to develop RandBLAS: a trustworthy and high-performance library for randomized dimension reduction (hereafter referred to as *sketching*). The second was the development of a novel randomized algorithm for computing a challenging type of matrix decomposition known as *Householder QR with column pivoting* (Householder QRCP).

1.1. RandBLAS

Our RandBLAS work built on a C++ prototype developed over two years in close coordination with maintainers of widely used NLA libraries. At the start of this LDRD, the prototype already offered performance and reproducibility that was unmatched within the ecosystem of RandNLA software. This LDRD resolved limitations of this prototype that impeded its deployment in DOE mission activities. Specifically, it added support for sparse data and created a statistical testing framework. These new features resulted in the first formal release of RandBLAS (version 0.2, in June 2024) and the first stable release of RandBLAS (version 1.0, in September 2024).

A case study: low-rank approximation of sparse matrices with truncated QRCP

RandBLAS's GitHub repository has several examples showing how it can be used to build high-level algorithms. Here we summarize one of those examples: low-rank approximation of sparse matrices via truncated QRCP. We choose this example to highlight two key points.

- RandBLAS provides implementations of the standard "SPMM" kernel for sparse-times-dense (or dense-times-sparse) matrix multiplication. This makes it possible to build high-level RandNLA algorithms for sparse matrix computations using only RandBLAS and an LAPACK-like library for classical dense matrix computations. See Algorithm 1 for details.
- RandBLAS' sparse matrix multiplication kernels are fast enough that they are not the bottleneck operation in the larger algorithm (see Figure 1-1). Therefore, while these kernels could be optimized even further, it makes more sense to focus future development work on integration with hardware accelerators and linear algebra libraries used in DOE HPC workflows.

Algorithm 1 Pseudo-code for a practical RandBLAS-powered randomized algorithm for truncated QRCP of sparse matrices. The basic idea of the randomized truncated QRCP algorithm is to make pivot decisions by looking at a small sketch of the data matrix. The origins of and variations on this approach to low-rank approximation can be found in [10, 19–21]. The same idea is at the heart of randomized algorithms for Householder QRCP [8, 12, 13, 22]. The end of each line shows the library functions needed for the stated operation.

Read-only arguments. Sparse $\mathbf{A} \in \mathbb{R}^{m\times n}$ in CSC format.

Read-write arguments. Buffers for $\mathbf{Q} \in \mathbb{R}^{m\times k}$, $\mathbf{R} \in \mathbb{R}^{k\times n}$, and $\boldsymbol{p} \in \mathbb{N}^n$.

1: **function** `truncated_sparse_qrcp`$(\mathbf{A}, \mathbf{Q}, \mathbf{R}, \boldsymbol{p}, \ell = 0)$
2: $\quad \ell_{\text{done}} = 0$
3: $\quad$ **if** ℓ is even **then**
4: $\quad\quad$ overwrite $\mathbf{Q}$ with a $k \times m$ iid Gaussian matrix — use `RandBLAS::fill_dense`
5: $\quad$ **else**
6: $\quad\quad$ overwrite $\mathbf{R}$ with a $k \times n$ iid Gaussian matrix — use `RandBLAS::fill_dense`
7: $\quad\quad$ overwrite $\mathbf{Q} = \mathbf{RA}$ — use `RandBLAS::spmm_right`
8: $\quad\quad$ orthogonalize the rows of $\mathbf{Q}$ — use `lapack::gelqf` and `unglq`
9: $\quad\quad$ $\ell_{\text{done}} += 1$
10: $\quad$ **while** $(\ell - \ell_{\text{done}} > 0)$ **do**
11: $\quad\quad$ overwrite $\mathbf{R} = \mathbf{QA}$ — use `RandBLAS::spmm_right`
12: $\quad\quad$ orthogonalize the rows of $\mathbf{R}$ — use `lapack::gelqf` and `unglq`
13: $\quad\quad$ overwrite $\mathbf{Q} = \mathbf{RA}^*$ — use `RandBLAS::spmm_right`
14: $\quad\quad$ orthogonalize the rows of $\mathbf{Q}$ — use `lapack::gelqf` and `unglq`
15: $\quad\quad$ $\ell_{\text{done}} += 2$
16: $\quad$ overwrite $\mathbf{R} = \mathbf{QA}$ — use `RandBLAS::spmm_right`
17: $\quad$ decompose $\mathbf{R}(:, \boldsymbol{p}) = \tilde{\mathbf{Q}}\tilde{\mathbf{R}}$ in-place — use `lapack::geqp3`
18: $\quad$ overwrite $\mathbf{Q}$ by $\mathbf{A}(:, \boldsymbol{p}_1), \ldots, \mathbf{A}(:, \boldsymbol{p}_k)$.
19: $\quad$ orthogonalize the columns of $\mathbf{Q}$ with preconditioned Cholesky QR.

- overwrite $\mathbf{Q} = \mathbf{Q}(\tilde{\mathbf{R}}(1{:}k, 1{:}k))^{-1}$ — use `blas::trsm`
- overwrite the first k^2 entries of $\mathbf{R}$ by the matrix $\mathbf{G} = \mathbf{Q}^*\mathbf{Q}$ — use `blas::syrk`
- decompose $\mathbf{G} = \mathbf{LL}^*$ in-place — use `blas::potrf`
- overwrite $\mathbf{Q} = \mathbf{Q}(\mathbf{L}^*)^{-1}$ — use `blas::trsm`

20: $\quad$ overwrite $\mathbf{R} = \mathbf{Q}^*\mathbf{A}$ — use `RandBLAS::spmm_right`
21: $\quad$ pivot $\mathbf{R} = \mathbf{R}(:, \boldsymbol{p})$ in-place
22: $\quad$ **return**

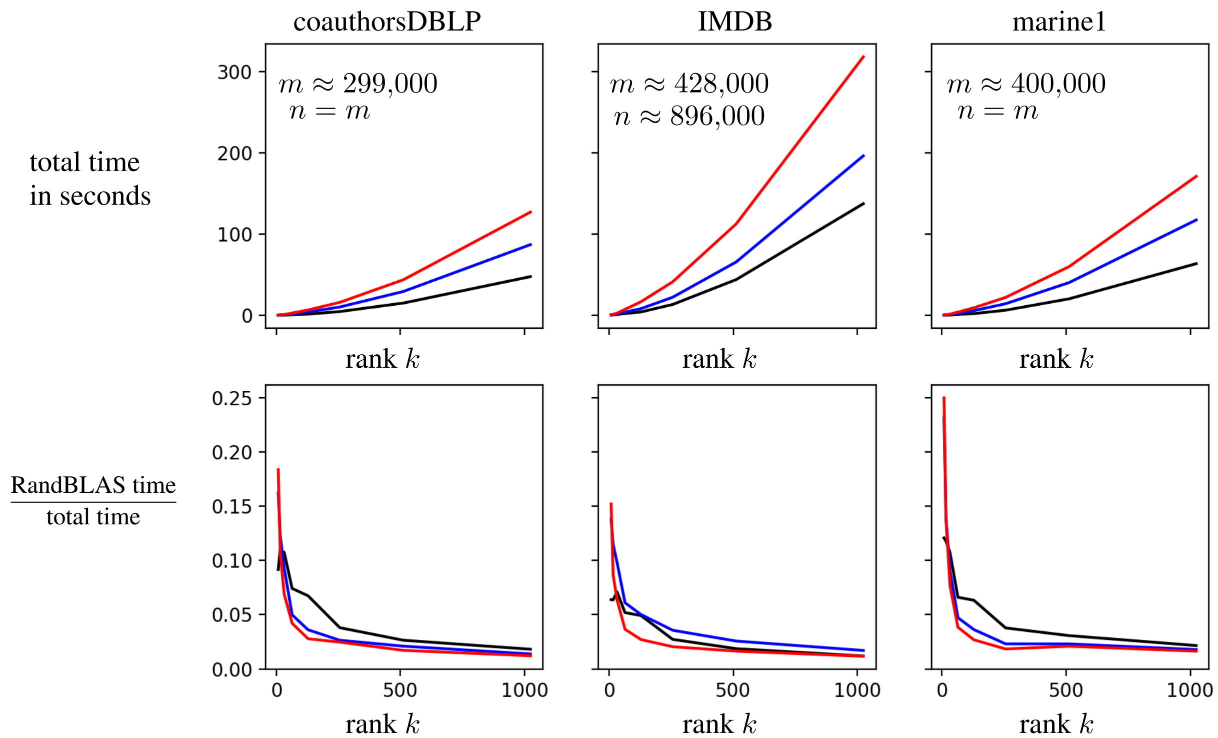


Figure 1-1. Runtime data for Algorithm 1 on sparse matrices in the SuiteSparse matrix collection, using an M2 Max MacBook Pro (12 cores, 48 MB L3 cache, 96 GB RAM) and running one OpenMP thread per core. Matrix dimensions are indicated with superimposed text. The black, blue, and red curves show data when $\ell = 0$, $\ell = 1$, and $\ell = 2$, respectively. Plots in the lower row show that algorithm runtime is dominated by BLAS and LAPACK functions rather than RandBLAS functions, and this becomes more pronounced as k increases.

1.2. Householder QRCP

Remark 1.2.1. In what follows, "QRCP" always means "Householder QRCP."

Our work on a faster QRCP algorithm was chosen because QRCP is a major pain-point in NLA computations; QRCP is a fundamental decomposition that nominally has the same complexity as unpivoted QR, and yet on larger matrices the standard algorithm can run 100 times slower than complexity analysis would predict. Our approach was to combine a recently developed algorithm for QRCP of very tall matrices with earlier ideas for randomized QRCP of general matrices. Our implementations of this algorithm show exceptional performance – on both CPUs and NVIDIA GPUs, they come within a factor 2 of the speed of unpivoted QR on large matrices.

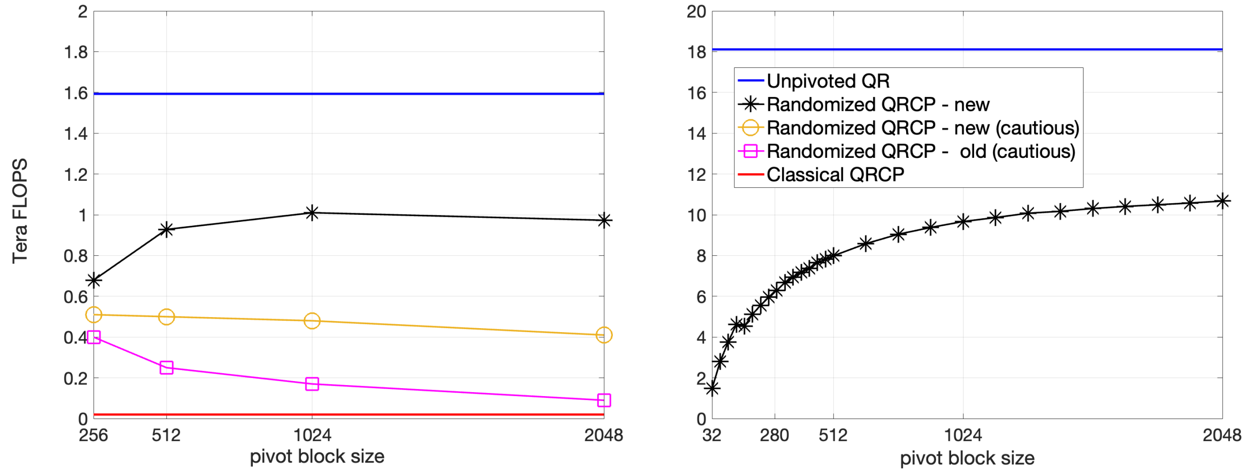


Figure 1-2. Standardized flop rates for QR algorithms applied to large square matrices on two architectures (computed by taking the flop count of unpivoted Householder QR and dividing by algorithm runtime). The x-axis shows different choices of block size for randomized algorithms. *Left*: CPU algorithms for square matrices of order 65536. *Right*: GPU algorithms for square matrices of order 32768 (NVIDIA H100). Only two algorithms were run on the H100 because cuSOLVER does not implement a QRCP algorithm.

		Intel Xeon Gold 6248R (2x)
Cores per socket		24
Clock Speed	**Base**	3.00 GHz
	Boost	4.00 GHz
Cache sizes per socket	**L1**	1536 KB
	L2	24 MB
	L3	35.75 MB
RAM	**Type**	DDR4-2933
	Size	192 GB
BLAS & LAPACK		MKL 2024.2
GCC & G++		10.2.0
CMake		3.23.2
OS		Red Hat Enterprise Linux 8.7

Table 1-1. System specifications for benchmarks in the left panel of Figure 1-2.

1.3. Products of this LDRD

This LDRD led to several papers, software products, and presentations, as outlined below. In addition to these "standard" products, the LDRD PI consulted extensively on an educational YouTube video on RandNLA, which now has over 292,000 views only five months after release.[1]

Indirect follow-on funding has been provided by Dr. Michael Mahoney (with UC Berkeley, LBNL, and the International Computer Science Institute), who hired a one-year post-bac to contribute to the RandNLA software developed in this LDRD.

Papers

Fast multiplication of random dense matrices with sparse matrices. IPDPS, 2024. T. Liang, R. Murray, J. Demmel, and A. Buluç.

Anatomy of high-performance QR with column pivoting. In preparation, writing led by collaborators at UT Knoxville.

Basic subroutines for randomized linear dimension reduction. In preparation, writing led by R. Murray.

Software

RandBLAS: `https://github.com/BallisticLA/RandBLAS`.

> We made the first formal release of RandBLAS and its first stable release. In August 2024 we had our first organic contributor (that is, someone who volunteered to contribute despite no connections to the project's origins). GitHub stars increased from 5 to 73.[2]

RandLAPACK: `https://github.com/BallisticLA/RandLAPACK`.

> We added our novel algorithm for QRCP, with both CPU and GPU implementations. GitHub stars increased from 7 to 57.
>
> We also added initial support for kernel methods, such as kernel ridge regression and kernel PCA, in `https://github.com/BallisticLA/RandLAPACK/pull/85`.

spaND: `https://github.com/leopoldcambier/spaND_public`.

> This is C++ code for solving sparse positive definite linear systems, originally developed with SNL LDRD funding [1, 3]. We modernized spaND and improved its portability while exploring applications of our QRCP work; see `https://github.com/leopoldcambier/spaND_public/pull/1`.

[1] https://www.youtube.com/watch?v=6htbyY3rH1w

[2] Stars are public bookmarks made by those in the software development community.

Presentations

The roles of sparse matrices in emerging standards for randomized dimension reduction. R. Murray. Sparse BLAS Workshop. University of Tennessee, Knoxville. November 2023.

Novel randomized algorithms for QR with column pivoting, and their implementations in RandLAPACK. M. Melnichenko. SIAM Conference on Parallel Processing. March 2024.

Fast low-rank QRCP of sparse matrices – with RandBLAS! R. Murray. SIAM Conference on Applied Linear Algebra. May 2024.

Novel randomized algorithms for QR with column pivoting, and their implementations in RandLAPACK. M. Melnichenko. SIAM Conference on Applied Linear Algebra. May 2024.

The frontier of randomization for scalable data analysis and matrix computations. R. Murray. Rice University, Statistics Departmental Seminar. September 2024.

1.4. Assessment and outlook

While the papers describing this LDRD's products have yet to be finished, this LDRD remains a significant success. The technical goals for Householder QRCP were achieved to a greater extent than we thought possible. Most technical goals for RandBLAS were also met – the exceptions being continued dependence on BLAS++ as a portability layer and the absence of wrappers for C or Rust. Those specific goals were not met because information learned very early in the LDRD indicated that good performance for sparse matrices was imperative and focus on C++ was justified by its prominence in Sandia' HPC software stack.

We're optimistic that our work on Householder QRCP will be received favorably by the broader HPC community, since it continues the tradition of dense linear algebra libraries emphasizing full matrix decompositions. Householder QRCP in particular is a valuable kernel for solving least squares problems, owing to its exceptionally robust numerical stability.

2. DISCUSSION OF TECHNICAL WORK ON RANDBLAS

As a matter of notation, we use $[\![k]\!]$ to denote the index set $\{0, \ldots, k-1\}$.

2.1. Preliminaries

2.1.1. Short and long axes of sketching operators

Sketching operators are only "useful" for dimension reduction if they're non-square. The larger dimension of a sketching operator has a different semantic role than the smaller dimension. To put this distinction front-and-center we use the term *short-axis vectors* in reference to the columns of a wide matrix or the rows of a tall matrix, and we use *long-axis vectors* in reference to the rows of a wide matrix or the columns of a tall matrix. The length of an operator's short axis is equal to what's commonly called "sketch size." (The literature has no common name for the length of a sketching operator's long axis. Terms like "input dimension" or "ambient dimension" might be appropriate.)

Remark 2.1.1. We refer to short and long axes programatically with RandBLAS' `Axis` enumeration, which has values `Axis::Short` and `Axis::Long`.

The concepts of short-axis and long-axis vectors are very important for RandBLAS' definitions of sparse sketching operators. Depending on whether the sparse operator is defined with respect to one of these axes or another will affect whether the operator acts as a linear hash function or as an average of coordinate sampling operators. See Figure 2-1 and Section 2.2.2 for more information.

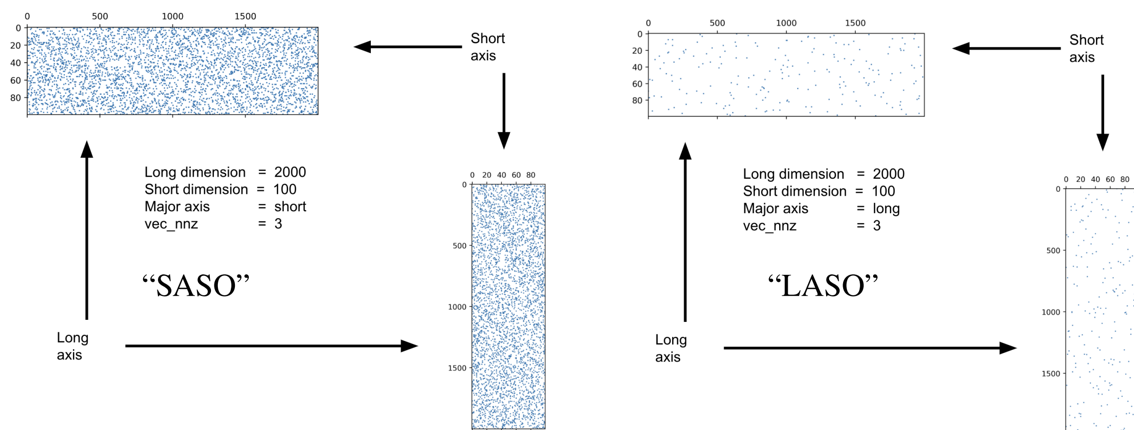


Figure 2-1. Sparsity patterns of short-axis-sparse and long-axis-sparse operators ("SASOs" and "LASOs"). Note that transposition does not affect whether an operator is a SASO or LASO.

2.1.2. Counter-based random number generators

A *counter-based random number generator* or *CBRNG* is a type of pseudo-random number generator based on principles described in [18]. We need some definitions in order to explain these principles.

First, let $\mathbb{U}_p$ denote the ring of unsigned p-bit integers, equipped with addition, multiplication, and standard (wrap-around) overflow rules. We identify $(\mathbb{U}_p)^q = \mathbb{U}_{pq}$ by reading in a block little-endian format. For example, if $c = (c_1, \ldots, c_4) \in (\mathbb{U}_p)^4$ with $c_i \in \mathbb{U}_p$, then

$$c \equiv c_1 + 2^p \cdot c_2 + 2^{2p} \cdot c_3 + 2^{3p} \cdot c_4. \tag{2.1.1}$$

These definitions in mind, a CBRNG is a family of *keyed bijections* on L-tuples of B-bit integers. A specific bijection $\Phi_k : \mathbb{U}_{LB} \to (\mathbb{U}_B)^L$ generates a pseudo-random stream

$$\begin{bmatrix}\Phi_k(0) & \Phi_k(1) & \cdots & \Phi_k(2^{BL}-2) & \Phi_k(2^{BL}-1)\end{bmatrix}. \tag{2.1.2}$$

This stream gives us $L2^{BL}$ integers in total; it's extended to all $c \in \mathbb{N}$ by defining $\Phi_k(c) = \Phi_k(c \bmod 2^{BL})$.

The original CBRNGs (i.e., those defined in [18]) were all implemented in the Random123 library. In the formal context of Random123's API, a CBRNG is is not a specific keyed bijection Φ_k, but rather a mapping $(k, c) \mapsto \Phi_k(c)$. This is done so Random123's CBRNGs are truly *stateless functions*.

2.2. Distributions, and how we sample from them

RandBLAS defines a type called `RNGState` that acts as a container for triplets (Φ, k, c) where Φ is a CBRNG and (k, c) are Random123-defined datatypes for a *key* and a *counter*. An `RNGState` is effectively a pointer to a location in a sequence of vectors of the form (2.1.2), along with rules for how to proceed along the stream. RandBLAS uses `RNGState` to facilitate lazily sampling of the data used in explicit representations of a sketching operator.

We distinguish between distributions over random matrices and samples from those distributions. The minimal properties of these formalisms are constrained by RandBLAS' `SketchingDistribution` and `SketchingOperator` C++20 concepts. A `SketchingOperator` is basically a pair of an `RNGState` and a `SketchingDistribution`.

RandBLAS has two types to represent distributions over random matrices: `DenseDist` and `SparseDist`. These types have members called `major_axis`, `dim_major`, and `dim_minor` that aren't required by the `SketchingDistribution` concept. The `major_axis` can be equal to `Axis::Long` or `Axis::Short`. We call a distribution *short-axis major* or *long-axis major* depending on this value. Given this and $(\texttt{n_rows}, \texttt{n_cols})$, we set

$$\texttt{dim_major} = \begin{cases} \min\{\texttt{n_rows},\ \texttt{n_cols}\} & \text{if } \texttt{major_axis} = \texttt{Axis::Short} \\ \max\{\texttt{n_rows},\ \texttt{n_cols}\} & \text{if } \texttt{major_axis} = \texttt{Axis::Long} \end{cases},$$

and we set `dim_minor` to whichever of $(\texttt{n_rows}, \texttt{n_cols})$ isn't identified as `dim_major`.

2.2.1. *Dense sketching operators*

The entries of dense sketching operators in RandBLAS are sampled iid from one of two mean-zero variance-one distributions: the standard normal distribution (indicated by `RandBLAS::ScalarDist::Gaussian`) or the uniform distribution over $[-\sqrt{3}, \sqrt{3}]$ (indicated by `RandBLAS::ScalarDist::Uniform`).

Our methods for sampling these operators *effectively* take in an `RNGState` and an integer $n =$ `dim_major`, use these to transform (2.1.2) into a stream of vectors in $\mathbb{R}^n$, and then stack $m =$ `dim_minor` consecutive random vectors from this stream as the rows or columns of a matrix. More formally ...

- We take in an `RNGState` `s` $= (\Phi, k, c)$. This defines a sequence of vectors

$$\Phi_k(c),\ \Phi_k(c+1),\ \Phi_k(c+2),\ \ldots \in (\mathbb{U}_B)^L.$$

- We map $(\mathbb{U}_B)^L$ to $\mathbb{R}^L$ using an appropriate distribution-specific transformation.

- We stack vectors from $\mathbb{R}^L$ to length $\geq n$, then truncate to first n components.

A sequence of m vectors constructed in this way is stacked to produce either an $m \times n$ matrix or an $n \times m$ matrix, depending on the relative sizes of (m, n) and whether the distribution is short-axis or long-axis major.

2.2.2. *Sparse sketching operators*

A SparseDist represents a distribution over sparse matrices with fixed dimensions, where either the rows or the columns are sampled independently from a certain distribution over sparse vectors. The distribution is determined by `major_axis` and a parameter called `vec_nnz`.

Let $k =$ `dim_major`. The major-axis vectors of a SparseSkOp follow a distribution $\mathcal{V}$ over $\mathbb{R}^k$. The number of nonzeros in each major-axis vector is bounded by $1 \leq$ `vec_nnz` $\leq k$.

All else equal, larger values of `vec_nnz` result in distributions that are "better" at preserving Euclidean geometry when sketching. The value of `vec_nnz` that suffices for a given context will also depend on the sketch size, $d := \min\{\texttt{n_rows}, \texttt{n_cols}\}$. Larger sketch sizes make it possible to "get away with" smaller values of `vec_nnz`.

2.2.2.1. SASOs: short-axis-sparse operators

A sample from $\mathcal{V}$ has exactly `vec_nnz` nonzeros. The locations of those nonzeros are chosen uniformly without replacement from $[\![k]\!]$. The values of the nonzeros are independent Rademachers.

Many sketching distributions from the literature fall into this category. `vec_nnz` $= 1$ corresponds to the distribution over CountSketch operators. `vec_nnz` > 1 corresponds to distributions which have been studied under many names, including OSNAPs [15], SJLTs [5], and hashing embeddings [4].

```
template <typename state_t>
inline void considerate_fisher_yates(
    state_t &state, int k, int n, int *samples, int *indices, int *pivots
) {
    for (int j = 0; j < k; ++j) {
        int p = random_uint(j, n, state.counter, state.key);
        pivots[j] = p;
        swap(indices[p], indices[j]);
        samples[j] = indices[j];
        state.counter.incr();
    }
    for (int j = 1; j <= k; ++j) {
        int i = k - j;
        int s = samples[i];
        int p = pivots[i];
        indices[i] = indices[p];
        indices[p] = s;
    }
    return;
}
```

Figure 2-2. Example function for sampling k elements from the length-n array "`indices`" uniformly without replacement, using k steps of Fisher-Yates shuffling. The first for-loop performs the sampling, using a black-box function called `random_unit` that accepts integers "`lo`" and "`hi`," and a CBRNG counter and key, and returns an unsigned integer sampled uniformly at random from `lo` to `hi - 1`. The second for-loop restores `indices` to the state it had on entry. RandBLAS does not use this exact code.

How to choose vec_nnz. The community has come to a consensus that very small values of vec_nnz can suffice for good performance. For example, suppose we seek a constant-distortion embedding of an unknown subspace of dimension n, where $1{,}000 \le n \le 10{,}000$. If $d = 2n$, then many practitioners would restrict their attention to vec_nnz ≤ 8. There are no special performance benefits in RandBLAS to setting vec_nnz $= 1$. Additionally, using vec_nnz > 1 makes it far more likely for a sketch to retain useful geometric information from the data. Therefore, we recommend using vec_nnz ≥ 2 in practice.

Efficient sampling. RandBLAS constructs SASOs with a specialized version of Fisher-Yates shuffling.

As background, Fisher-Yates shuffling is a method for uniformly sampling permutations of $[\![d]\!]$. The method is iterative, and can be stopped after ℓ steps to obtain a ℓ elements chosen uniformly without replacement from $[\![d]\!]$, which is what we need for SASOs.

The bottleneck operation in an ℓ-step Fisher-Yates shuffle of $[\![d]\!]$ is preparing an explicit (array) representation of $[\![d]\!]$. We usually need to make $m \gg d$ successive calls to such a sampling algorithm. While this would naively cost $\Theta(md)$ operations, the cost of preparing the explicit representation of $[\![d]\!]$ can be amortized. RandBLAS' method for amortizing the cost is to track additional information in the Fisher-Yates shuffle so that the algorithm can be run in reverse and restore input workspace to the same state it had on entry. The resulting method runs in time $\Theta(d + m\ell)$; see Figures 2-2 and 2-3.

```
template <typename state_t>
static state_t repeated_fisher_yates(
    const state_t &state, int k, int n, int r, int *samples
) {
    int *pivots  = new int[k];
    int *indices = new int[n];
    for (int j = 0; j < n; ++j)
        indices[j] = j;

    auto out = state;
    for (int i = 0; i < r; ++i) {
        considerate_fisher_yates(
            out, k, n, samples + i*k, indices, pivots
        );
    }
    delete [] pivots;
    delete [] indices;
    return out;
}
```

Figure 2-3. Example function for making r independent samples of k elements from $[\![n]\!]$ uniformly without replacement. The function's total runtime is $\Theta(kr + n)$. RandBLAS does not use this exact code.

2.2.2.2. LASOs: long-axis-sparse operators

A sample $\boldsymbol{x}$ from $\mathcal{V}$ has *at most* `vec_nnz` nonzero entries. The locations of the nonzeros are determined by sampling uniformly with replacement from $[\![k]\!]$. If index j occurs in the sample ℓ times, then $\boldsymbol{x}_j$ will equal $\sqrt{\ell}$ with probability 1/2 and $-\sqrt{\ell}$ with probability 1/2.

In the literature, `vec_nnz` $= 1$ corresponds to operators for sampling uniformly with replacement from the rows or columns of a data matrix (although the signs on the rows or columns may be flipped). Taking `vec_nnz` > 1 gives a special case of LESS-uniform distributions [6], where the underlying scalar sub-gaussian distribution is the Rademacher distribution.

It is important to use (much) larger values of `vec_nnz` with LASOs compared to with SASOs, at least for the same sketch size d. There is less consensus in the community for what constitutes "big enough in practice," therefore we make no prescriptions on this front.

Sampling uniformly with replacement from an index set is very simple. If the sampling is performed sequentially then it's easy to scale the nonzero values in the way required by the definition of $\mathcal{V}$. If the sampling is performed in parallel (or via any black-box) then separate logic is needed to merge repeated indices within each major-axis vector. See Figure 2-4 for a straightforward way of doing this.

```
template <typename T = double>
int laso_merge_long_axis_vector_repeats(
    int k, T* vals, int* locs, std::unordered_map<int, T> &loc2count
) {
    loc2count.clear();
    for (int j = 0; j < k; ++j) {
        int i = locs[j];
        T val = vals[j];
        if (loc2count.count(i)) {
            loc2count[i] = loc2count[i] + 1;
        } else {
            loc2count[i] = 1.0;
        }
    }
    int num_unique_locs = (int) loc2count.size();
    if (num_unique_locs < k) {
        // Then we have duplicates. We need to overwrite some of the values
        // of locs and vals. Entries of these arrays at or past index
        // "loc2count.size()" will be ignored by the calling function.
        int i = 0;
        for (const auto& [j,c] : loc2count) {
            locs[i] = j;
            vals[i] *= std::sqrt(c);
            i += 1;
        }
    }
    return num_unique_locs;
}
```

Figure 2-4. `vals` and `locs` are length-k buffers; the former contains independent samples from a mean-zero variance-one sub-gaussian distribution (like the Rademacher distribution) and the latter contains integers sampled uniformly with replacement from some index set. On exit, the first `num_unique_locs` entries of `locs` have been overwritten with the unique values that `locs` had on entry, and `vals[j]` is multiplied by the square root of the number of times that `locs[j]` occurred in `locs`. RandBLAS does not use this exact code.

2.3. Basic statistical tests

RandBLAS gets its CBRNGs from Random123. Our tests include *known answer tests* from Random123's test suite, and tests which verify that Random123's counters behave in the way described in (2.1.1). Here we outline some of RandBLAS' distributional tests.

2.3.1. Background: Kolmogorov-Smirnov

Let F_X be the empirical CDF generated by iid samples $X_1, \ldots, X_N$ from a distribution $\mathcal{D}$ on $\mathbb{R}$. The Kolmogorov-Smirnov (KS) test statistic comparing F_X to a model F is

$$\mathrm{KS}_X = \|F_X - F\|_{\sup}. \tag{2.3.1}$$

The null hypothesis of the KS test is that F is the CDF of $\mathcal{D}$. We reject the null hypothesis up to some significance level α if KS_X exceeds a critical value $C_{N,\alpha}$ obtained from precomputed statistical tables.

The KS test has two value propositions for our purposes. First, it's nonparametric (hence broadly applicable). Second, its interpretation is self-evident; the test statistic is simply a sup-norm distance between the expected value and an observed value.

2.3.2. Sampling from index sets

The KS test is easy to implement for code that purports to sample from an index set $[\![n]\!]$ according to a probability mass function $\boldsymbol{p} \in \mathbb{R}^n$. We have separate tests for the uniform and nonuniform distributions, where the sampling is handled by `sample_indices_iid_uniform` and `sample_indices_iid`, respectively. In both cases we consider significance levels $\alpha \in \{10^{-6}, 10^{-4}, 10^{-2}\}$ and three combinations of index set size and sample size:

$$(n, N) \in \left\{(10^2, 10^5),\ (10^4, 10^3),\ (10^6, 10^3)\right\}.$$

For the nonuniform sampling we consider when $p_i = w_i / \sum_j w_j$, after initializing $w_j = 1/(1+i)^q$ for an exponent $q = 1$ (in some tests) or $q = 3$ (in others).

Our tests for nonuniform sampling also consider two generate distributions when $n = 100$.

- In one case, $\boldsymbol{p}$ is obtained by initializing a vector $\boldsymbol{w} = \mathbf{0}$, setting $w_i = 1/(1+i)$ for even values of $i \neq 10$, and then normalizing $\boldsymbol{p} = \boldsymbol{w}/\sum_i w_i$. In addition to running Fisher–Yates as a statistical test, we check that neither the number 10 and nor any odd numbers are produced by `sample_indices_iid`.
- In another case, $\boldsymbol{p}$ is a delta function: $p_{17} = 1$. Here no KS test is necessary; we simply check that all samples produced by `sample_indices_iid` are equal to 17.

Sampling uniformly from an index set without replacement (handled by `repeated_fisher_yates`) does not lend itself to direct testing by KS. Still, we can construct a test for our main use-case of this function: constructing the columns of a wide SASO $\mathbf{S}$. The idea is that we can look at the off-diagonal entries of $\mathbf{G} = \mathbf{S}^*\mathbf{S}$. It's easy to see that the entries within a single row or column of $\mathbf{G}$ are independent. Furthermore, an individual off-diagonal entry G_{ij} is distributed as a Rademacher series of random length. The number of terms in the Rademacher series is simply the number of shared nonzero index locations for columns i and j of $\mathbf{S}$. If $\mathbf{S}$ is $d \times m$ and has k nonzeros per column, then the number of nonzero index locations shared by any distinct pair of columns should follow the hypergeometric distribution where k draws are taken from a set with k distinguished elements and $d - k$ 'unremarkable' elements.

2.3.3. Continuous distributions on the reals

We use KS to check RandBLAS' functions for sampling from the uniform distribution over $[-\sqrt{3}, \sqrt{3}]$ and the standard normal distribution. There are complications in evaluating (2.3.1) here because the empirical CDF will always be piecewise constant, while the model CDFs are continuous. Figure 2-5 shows how the evaluation can be done correctly.

```cpp
template <typename T = double>
static T continuous_ks_evaluator(std::vector<T> &X, ScalarDist sd) {
    // First, define a function handle for the model CDF.
    auto F_model = [sd](T t) {
        if (sd == ScalarDist::Gaussian) {
            return standard_normal_cdf(t);
        } else {
            return uniform_symmetric_interval_cdf(t, (T) sqrt(3));
        }
    };
    auto N = (int) X.size();
    /**
     *  Let L(t) = |F_X(t) - F_model(t)|. The KS test testatistic is
     *
     *      ts = sup_{all t} L(t).
     *
     *  Now set s = sorted(X), and partition the real line into
     *
     *      I_0     = (-infty,  s[0  ]),  ...
     *      I_1     = [s[0  ],  s[1  ]),  ...
     *      I_2     = [s[1  ],  s[2  ]),  ...
     *      I_{N-1} = [s[N-2],  s[N-1]),  ...
     *      I_N     = [s[N-1],  +infty).
     *
     *  Then, provided F_model is continuous, we have
     *
     *      sup{ L(t) : t in I_j } = max{
     *              |F_model(inf(I_j)) - j/N|, |F_model(sup(I_j)) - j/N|
     *      }
     *
     *  for j = 0, ..., N.
     */
    X.push_back( numeric_limits<T>::infinity());
    X.push_back(-numeric_limits<T>::infinity());
    std::sort(X.begin(), X.end(), [](T a, T b) {return (a < b);});
    T test_statistic = 0.0;
    for (int j = 0; j <= N; ++j) {
        T empirical = (T)j / (T)N;
        T val1 = abs(F_model(X[j    ]) - empirical);
        T val2 = abs(F_model(X[j + 1]) - empirical);
        T supLt_on_Ij  = max(val1, val2);
        test_statistic = max(test_statistic, supLt_on_Ij);
    }
    return test_statistic;
}
```

Figure 2-5. Example showing how the Kolmogorov-Smirnov test statistic can be evaluated for continuous distributions in standard in C++. RandBLAS does not use this exact code.

2.4. Oblivious subspace embedding properties, and how to test for them

We call $\mathbf{S}$ a δ-*embedding* for the range of an orthonormal matrix $\mathbf{Q}$ if

$$1 - \delta \leq \sigma_{\min}(\mathbf{SQ}) \leq \sigma_{\max}(\mathbf{SQ}) \leq 1 + \delta.$$

Note that being a δ-embedding implies $\mathrm{cond}(\mathbf{SQ}) \leq (1+\delta)/(1-\delta)$.

Lemma 2.4.1 (Explicit in [17] when $k = n$; see also [7, 19] and [14])**.** *Let $\mathbf{A}$ be a rank-k matrix in $\mathbb{R}^{m \times n}$ with $m \geq n$. Suppose $\mathbf{S}$ is a $d \times m$ matrix $(d < m)$ and $\mathbf{P}$ is an $n \times k$ matrix where $\mathbf{Q} = \mathbf{SAP}$ is orthonormal.[1] The preconditioned matrix $\mathbf{AP}$ satisfies*

$$\mathrm{cond}(\mathbf{AP}) = \mathrm{cond}(\mathbf{SV})$$

where $\mathbf{V}$ is any orthonormal matrix with the same range as $\mathbf{A}$.

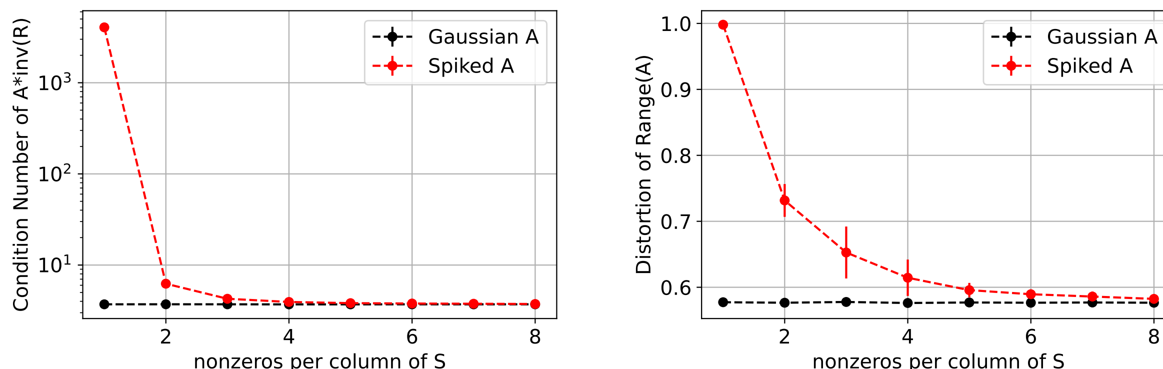


Figure 2-6. Visualizations of sketch quality for a wide $d \times m$ SASO for two types of $m \times n$ data matrices, where $(d, m, n) = (6000, 100000, 2000)$. The "Gaussian" matrix has iid standard-normal entries, while the "spiked" matrix is formed by stacking copies of the identity matrix and randomly scaling n rows by 10000. Plots show mean quantities computed with 10 different SASOs for a fixed matrix. Vertical lines in the right plot show sample standard deviations. The plots have two major messages. First, some data is "easier to sketch" than others. Second, even very small values of `vec_nnz` suffice to reduce distortion to very low levels.

Definition 2.4.1. Let Ω be a set of linear spaces of $\mathbb{R}^m$ and let $\mathcal{D}$ be a distribution over $d \times m$ matrices. We say that $\mathcal{D}$ is an *oblivious* (δ, p)-*embedding* for Ω) if

$$\Pr\{\mathbf{S} \sim \mathcal{D} \text{ is a } \delta\text{-embedding for } U\} \geq 1 - p \quad \text{for all} \quad U \text{ in } \Omega. \tag{2.4.1}$$

Definition 2.4.2. A distribution $\mathcal{D}$ on $\mathbb{R}^{d \times m}$ has the (n, δ, p)-*oblivious subspace embedding* (OSE) *property* if it's an oblivious (δ, p)-embedding for the set of all n-dimensional linear subspaces in $\mathbb{R}^m$.

It is desirable to test the subspace embedding properties of sketching operators supported by RandBLAS. This is difficult for two reasons. First, the theory available for sketching distributions is typically asymptotic, often

[1] For example, $\mathbf{P} = \mathbf{R}^{-1}$ from a QR decomposition of $\mathbf{SA}$.

suppressing constant factors and sometimes suppressing polylogarithmic factors. Second, it is expensive to estimate the probability that a sample from a sketching distribution will provide a δ-embedding for even *one* subspace of some modest dimension, let alone compute that probability for *all* subspaces of that dimension. However, these challenges are less severe for the Gaussian distribution, where theoretical results are known to be sharp (even in their constant factors) and where rotational invariance can be used to restrict our attention to coordinate subspaces.

Corollary 2.4.2 ([9]). *Let $\mathcal{G}_{d,m}$ denote the distribution over $d \times m$ matrices with i.i.d. $\mathcal{N}(0, 1/d)$ entries. Fix δ, p in $(0, 1)$. If n and d satisfy the following bounds for some $\tau > 0$:*

$$n \geq \left(\frac{\sqrt{2\log(1/p)} + 1}{\tau} \right)^2$$
$$d \geq n \left(\frac{1+\tau}{\delta} \right)^2$$

then $\mathcal{G}_{d,m}$ is an (n, δ, p)-OSE for any $m \geq n$.

By using Corollary 2.4.2, we can sweep over parameters δ and τ for any probability p of our choosing, and test only the lowest-dimensional subspaces for which the Gaussian distribution's OSE property should hold. Restricting the size of the test also has a major benefit of reducing complexity (both in the sense of "computational complexity" and "how complicated something is") of tests for the subspace embedding property. In particular, at small scales we can get away with the using the power method on $\mathbf{G} = (\mathbf{SQ})^*(\mathbf{SQ})$ and on $\mathbf{G}^{-1}$ to compute the extreme singular values of $\mathbf{SQ}$. This is valuable because it helps RandBLAS avoid a heavy dependency on a library like LAPACK in order to run its test suite. Our implementation sets the tolerance and number of iterations in the power method based on the following result.

Corollary 2.4.3 (Adaptation of Theorem 4.2, [11]). *Let $\mathbf{H}$ denote a nonzero positive semidefinite matrix of order n and let $\boldsymbol{x}_0$ denote a vector sampled uniformly at random from the unit sphere in $\mathbb{R}^n$. With probability one, the sequence of values $\lambda_k = \boldsymbol{x}_k^* \mathbf{H} \boldsymbol{x}_k$ generated by $\boldsymbol{x}_k = \mathbf{H}\boldsymbol{x}_{k-1}/\|\mathbf{H}\boldsymbol{x}_{k-1}\|_2$ is well-defined for all $k \geq 1$, and it converges to $\|\mathbf{H}\|_2$. Furthermore, for any ϵ and p in $(0, 1)$, setting k according to*

$$k \geq \min \left\{ \log\left(e + \frac{1-\epsilon}{\epsilon p^2} \log\left(\frac{1}{1-\epsilon} \right) \right),\ \log\left(\frac{\sqrt{n}}{p} \right) \right\} \Big/ \log\left(\frac{1}{1-\epsilon} \right)$$

ensures the probability bound $\Pr\{\lambda_k \geq (1-\epsilon)\|\mathbf{H}\|_2\} \geq 1 - p$.

2.5. Sparse matrix datastructures

We have minimalist implementations of CSC, CSR, and COO-format sparse matrices, via the `CSCMatrix`, `CSRMatrix`, and `COOMatrix` types. These types have no built-in support for accessing individual elements or for extracting submatrices. We expect that those who want to use RandBLAS for sparse matrix computations will already have a preferred library for working with sparse matrices, like Eigen, Armadillo, Kokkos, or MKL. The purpose of RandBLAS' sparse matrix classes is to provide lightweight views in the three major formats, around which we build our high-performance parallel implementations of sparse-times-dense (or dense-times-sparse) matrix multiplication.

Remark 2.5.1. SparseSkOp objects can own representations of their data in COO format. Our implementation of sparse sketching ultimately works by creating a COOMatrix view of an explicitly sampled SparseSkOp.

2.6. Multiplying sparse and dense matrices

There are three function names in RandBLAS' public API that involve multiplying sparse matrices: `spmm`, `sketch_general` (when used with sparse sketching operators), and `sketch_sparse`. These names are overloaded to handle when an operator of one type or another appears on the left or the right of the matrix-matrix product. Each overloaded function is marked as `inline` and dispatches a function with a (usually) more opaque name and less sophisticated templating.

2.6.1. Routing for `spmm`

The `spmm` overloads immediately route to `left_spmm` or `right_spmm`. Implementations of functions with any of these three names are contained in

`RandBLAS/sparse_data/spmm_dispatch.hh`.

`right_spmm` is implemented by falling back on `left_spmm` with transformed values for `layout` and for the transposition flags `opA` and `opB`. Here's what happens if `left_spmm` is called with a sparse matrix `A`, a dense input matrix `B`, and a dense output matrix `C`.

1. If needed, transposition of `A` is resolved by creating a lightweight object for the transpose called `At`. This object is just a tool for us to change how we interpret the buffers that underlie `A`.

    - If `A` is COO, then `At` will also be COO.
    - If `A` is CSR, then `At` will be CSC.
    - If `A` is CSC, then `At` will be CSR.

    We make a recursive call to `left_spmm` once we have our hands on `At`, so the rest of `left_spmm`'s logic only needs to handle un-transposed `A`.

2. A memory layout is determined for how we'll read `B` in the low-level sparse matrix multiplication kernels.

    - If `B` is un-transposed then we'll use the same layout as `C`.
    - If `B` is transposed then we'll swap its declared dimensions (i.e., we'll swap its reported numbers of rows and columns) and and we'll tell the kernel to apply it as an untransposed matrix stored in the opposite layout as `C`.

3. We dispatch a kernel from `coo_spmm_impl.hh`, or `csc_spmm_impl.hh`, or `csr_spmm_impl.hh`. The precise kernel we dispatch depends on the type of `A`, and the inferred layout for `B`, and the declared layout for `C`.

2.6.2. *Routing for* `sketch_general` *with* `SparseSkOp` *objects*

The `sketch_general` functions are defined in `skge.hh`. If we call one of these functions with a SparseSkOp object, `S`, we'd immediately get routed to either `lskges` or `rskges`. Here's what would happen after we entered one of those functions:

1. If the defining data `S` has yet to be sampled, then we create a shallow memory-owning copy of `S` and call `fill_sparse` on that shallow copy; the shallow copy frees its memory at destruction time.
2. We'd obtain a lightweight view of `S` as a COOMatrix, and we'd pass that matrix to `left_spmm` (if inside `lskges`) or `right_spmm` (if inside `rskges`).

2.6.3. *Routing for* `sketch_sparse`

If we call `sketch_sparse` with a DenseSkOp, `S`, and a sparse matrix, `A`, then we'll get routed to either `lsksp3` or `rsksp3`. From there, we'll do the following.

1. If the defining data `S` has yet to be sampled, then we instantiate an explicit representation of the relevant submatrix of `S` as a `BLASFriendlyOperator` (a type that's not in the public API) using `fill_dense_unpacked`. The memory required for this representation is freed at destruction time. We return after a recursive call to the current function with the `BLASFriendlyOperator`.
2. Remaining function logic assumes that the argument is either a DenseSkOp whose buffer representation is available *or* a `BLASFriendlyOperator` (which, by definition, has an explicit buffer representation). Using this buffer representation, we call ...

   - `right_spmm` if we're inside `lsksp3`.
   - `left_spmm` if we're inside `rsksp3`.

   Note that `l` and `r` in the names `[l/r]sksp3` get matched to opposite sides for `[left/right]_spmm`! This is because all the fancy abstractions in `S` have been stripped away by this point in the call sequence, so the "side" that we emphasize in function names changes from emphasizing `S` to emphasizing `A`.

3. DISCUSSION OF TECHNICAL WORK ON QRCP

Work summarized here was conducted in collaboration with Maksim Melnichenko and Mark Gates (University of Tennessee, Knoxville), James Demmel and Michael Mahoney (UC Berkeley and LBNL), Piotr Luszczek (MIT Lincoln Lab), and William Killian (Voltron Data; previously faculty at Millersville University). Details on this work will be shared in a forthcoming paper.

In this chapter, $[\![n]\!] = \{1, \ldots, n\}$.

3.1. Problem statement and background

Let $\mathbf{M}$ be a matrix of size $m \times n$ with at least as many rows as columns ($m \geq n$). QRCP is concerned with finding a permutation of $[\![n]\!]$ – call it J – along with a QR decomposition of the pivoted matrix $\mathbf{M}(:, J) = \mathbf{QR}$, such that leading and trailing singular values of $\mathbf{M}$ can be inferred from the spectra of leading and trailing blocks in 2×2 partitions of $\mathbf{R}$.[1]

QRCP is considerably more expensive than unpivoted QR from a communication standpoint, even with straightforward pivoting strategies. For example, Householder QR with Businger and Golub's *max-norm pivoting* requires updating column norms of every partial decomposition of $\mathbf{M}$ as the algorithm progresses from left to right across the columns [2]. These column-norm updates entail BLAS 2 operations, which are less suitable for modern hardware than the matrix-matrix (i.e., BLAS 3) operations abundant in classic algorithms for unpivoted QR. The significance of this problem has been known for decades [16].

Remark 3.1.1. The standard LAPACK function for Householder QR is `GEQRF`; the standard function for Householder QRCP (with max-norm pivoting) is `GEQP3`.

A significant amount of effort was devoted to improving QRCP's efficiency with randomization in the mid 2010's, beginning with independent works by Martinsson [12] and Duersch and Gu [8], and with subsequent extensions by Martinsson *et al.* [13] and Xiao, Gu, and Langou [22]. These randomized methods showed compelling performance at the time, but they were never adopted into standard linear algebra libraries. (This includes LAPACK, ScaLAPACK, SLATE, Eigen, Armadillo, Intel MKL, AMD AOCL, Arm Performance Libraries, IBM ESSL, Apple Accelerate, cuSOLVER, and MAGMA.)

One explanation for the lack of uptake of randomized QRCP is that available high-performance implementations of these methods did not adapt well to improvements in hardware. For example, experiments with the "HQRRP" code from [13] used an Intel CPU with the Sandy Bridge architecture (launched Q1'12), on which there was a 10x speed difference between `GEQRF` and `GEQP3`. However, when we compared `GEQRF` and `GEQP3` on Intel CPUs with the Cascade Lake architecture (launched Q1'20) we observed speed differences of 100x. While HQRRP did exhibit a nontrivial speedup over `GEQP3` on the newer architecture, it did not even come close to the performance of `GEQRF`.

[1] We use the term *spectrum* in reference to singular values, not eigenvalues. Eigenvalues are of interest to us only insofar as they coincide with singular values for positive semidefinite matrices.

This purpose of this LDRD's work on QRCP was to develop improved versions of these randomized algorithms along with high-performance implementations, with the aim of showing that RandNLA has major benefits to offer to the celebrated *decompositional approach to matrix computations*.

3.2. The high-level algorithm

The pseudocode Algorithm 2 represents the simplest formulation of our proposed algorithm. It's deliberately presented very similarly to [8, Algorithm 4].

Algorithm 2 Iterative CholeskyQR with randomization and pivoting

Required inputs. An $m \times n$ matrix M. An integer block size parameter b.

Optional inputs. A scalar γ that sets the size of the sketch relative to b ($\gamma \geq 1$).

Outputs. Orthogonal $\mathsf{Q} \in \mathbb{R}^{m\times k}$, upper-triangular $\mathsf{R} \in \mathbb{R}^{k\times n}$, and a permutation matrix $\mathsf{P} \in \mathbb{R}^{n\times n}$.

1: **function** $\texttt{icqrrp}(\mathsf{M}, b, \gamma = 1.25)$
2: Sample a Gaussian matrix S of dimensions $\lceil \gamma b \rceil \times m$
3: Allocate $\mathsf{Q}, \mathsf{R}, \mathsf{P}$
4: Sketch $\mathsf{M}^{\mathrm{sk}} = \mathsf{SM}$
5: **for** $i = 1 : n/b$ **do**
6: Decompose $[\sim, \hat{\mathsf{R}}, \hat{\mathsf{P}}] = \texttt{qrcp}(\mathsf{M}^{\mathrm{sk}})$
^ $\hat{\mathsf{P}} \in \mathbb{R}^{n\times n}$ is a permutation matrix
7: Determine $k = \texttt{rank}(\hat{\mathsf{R}})$
^ Performed by finding the last nonzero element in the diagonal of $\hat{\mathsf{R}}$
8: Permute $\mathsf{R}(:(i-1)b, (i-1)b:) = \mathsf{R}(:(i-1)b, (i-1)b:)\hat{\mathsf{P}}$
^ Only the rectangular portion of the computed rows is permuted
9: Truncate and precondition $\mathsf{M}^{\mathrm{pre}} = \mathsf{M}\hat{\mathsf{P}}(:, 1:k)(\hat{\mathsf{R}}(1{:}k, 1{:}k))^{-1}$
10: Decompose $[\mathsf{Q}^{\mathrm{econ}}, \mathsf{R}^{\mathrm{pre}}] = \texttt{cholqr}(\mathsf{M}^{\mathrm{pre}})$
^ $\mathsf{Q}^{\mathrm{econ}}$ is $m \times k$ and $\mathsf{R}^{\mathrm{pre}}$ is $k \times k$
11: Reconstruct $[\mathsf{Q}^{\mathrm{full}}, D] = \texttt{householder_reconstruct}(\mathsf{Q}^{\mathrm{econ}})$
^ D is a sign vector
12: Compute $\mathsf{R}^{11} = \mathsf{R}^{\mathrm{pre}}\texttt{diag}(D)\hat{\mathsf{R}}(1{:}k, 1{:}k)$
13: Compute $\mathsf{R}^{12} = \mathsf{Q}^{\mathrm{full}}(:, :b)^*\mathsf{M}\hat{\mathsf{P}}(:, (b+1):)$
14: Update $\mathsf{M} = \mathsf{Q}^{\mathrm{full}}(:, (b+1):)^*\mathsf{M}\hat{\mathsf{P}}(:, (b+1):)$
15: Update $\mathsf{R}((i-1)b : ib, (i-1)b:) = [\mathsf{R}^{11}\ \mathsf{R}^{12}]$
16: Update $\mathsf{Q} = \mathsf{Q}\mathsf{Q}^{\mathrm{full}}$
17: Update $\mathsf{P} = \mathsf{P}\hat{\mathsf{P}}$
18: **if** $i == n/b$ or $k \neq b$ **then**
break;
19: Update $\mathsf{M}^{\mathrm{sk}} = \begin{bmatrix} \hat{\mathsf{R}}^{12} - \hat{\mathsf{R}}^{11}(\mathsf{R}^{11})^{-1}\mathsf{R}^{12} \\ \hat{\mathsf{R}}^{22} \end{bmatrix}$
20: **return** $\mathsf{Q}, \mathsf{R}, \mathsf{P}$

Algorithm 2 stands out from its predecessors through employing *Cholesky QR* at step 10, followed by the use of `householder_reconstruct` function to restore the full representation of the $\mathbf{Q}$-factor, defined at a given iteration. On its own, this is a very minor innovation. The contribution of this LDRD research is to elucidate design patterns for high-performance implementation of randomized QRCP algorithms.

3.2.1. A fast QRCP, suitable for wide sketches

Step 6 in pseudocode Algorithm 2 uses a black-box `qrcp` function, applied to a wide matrix $\mathbf{M}^{\text{sk}}$. By default, one could choose to use the standard LAPACK QRCP function, `GEQP3`, for the lack of an immediate alternative. In Algorithm 3, we show a much faster approach that utilizes LAPACK's row-pivoted LU function, `GETRF` to retrieve the pivot vector $\hat{J}$ (an alternative data view to the pivot matrix $\hat{\mathbf{P}}$ in step 6) and then unpivoted QR with `GEQRF` to find the matrix $\hat{\mathbf{R}}$ (a portion of which will be used for preconditioning).

Algorithm 3 : `qrcp_practical`

Input: A matrix $\mathbf{M} \in \mathbb{R}^{d \times n}$, where $d \ll n$

1: **function** `qrcp_practical`($\mathbf{M}$)
2: Allocate $\mathbf{M}^{\text{trans}} \leftarrow$ `transposition`($\mathbf{M}$)
3: Compute $[\sim, \sim, J_{\text{lu}}] =$ `lu`($\mathbf{M}^{\text{trans}}$)
 # ^ Done via standard row-pivoted LU factorization, `GETRF`
4: Convert $J_{\text{qr}} =$ `piv_transform`(J_{lu})
5: Compute $[\sim, \mathbf{R}] =$ `qr`($\mathbf{M}(:, J)$)
 # ^ Done via standard unpivoted QR factorization, `GEQRF`
6: **return** $\mathbf{R}$, $\mathbf{J}$

Step 2 in Algorithm 3 *does not* use an in-place transpose. Since the matrix $\mathbf{M}^{\text{sk}}$ that is to be input into `qrcp` function in step 3 is substantially smaller than $\mathbf{M}$, allocating a $d \times n$ buffer for $\mathbf{M}^{\text{trans}}$ is more reasonable than performing an additional in-place transpose to restore $\mathbf{M}^{\text{sk}}$ at the end of the algorithm.

Step 4 in Algorithm 3 is crucial, since the format in which a pivoted LU represents the permutation vector J is different from the pivoted QR format. In the context of pivoted LU factorization, row i of the input matrix was interchanged with row $J_{\text{lu}}[i]$. With that, the conversion procedure consists of first creating a vector J_{qr} of length n with entries from 1 to n and then serially swapping elements in it in accordance with the entries in J_{lu}. Simply put, for element at index $i = 1, \dots n$, element at $J_{\text{qr}}[i]$ is to swap positions with the element at $J_{\text{qr}}[J_{\text{lu}}[i]]$. It is important to remember that pivot vectors in LAPACK usually store entries in a one-based index format.

It is important to note that using Algorithm 3 has additional implications for the process of sketching. First, when Algorithm 3 is in use, there is no reason to set γ to anything other than 1, since GETRF's first k pivots for an input matrix only depend on the first k rows of that matrix. Second, using a *sparse* sketching operator in combination with Algorithm 3 may result in a failure on due to the pivoting rule in a pivoted LU function. This, however, is not an issue, as there is no performance benefit to be had by using a sparse sketching operator instead of a Gaussian operator in our context.

3.2.2. Column permutations

As part of developing a high-performance QRCP algorithm, we needed to write a conceptually trivial, yet crucially important kernel - a function for permuting columns of a given matrix in accordance with the output

of `qrcp` at step 6 of Algorithm 2. In the context of pivoted QR factorization that outputs a pivot vector J_{qr} if $J_{\text{qr}}[j] = i$, then the jth column of $\mathbf{A}(:, J_{\text{qr}})$ was the kth column of $\mathbf{A}$; this representation of the pivot vector can be referred to as "permutation format." A simple method for applying the column permutation is given in Algorithm 4.

Algorithm 4 : `col_perm_sequential`

Input: A matrix $\mathbf{M} \in \mathbb{R}^{d \times n}$, where $d \ll n$, a pivot vector J_{qr} output by a black-box `qrcp` function, and integer $\ell \leq n$ describing the number of columns to be swapped.

1: **function** `col_perm_sequential`$(\mathbf{M}, J_{\text{qr}}, \ell)$
2: **for** $i = 1 : \ell$ **do**
3: $j = J_{\text{qr}}[i]$
4: `swap`$(\mathbf{M}[:, i], \mathbf{M}[:, j])$
^ Swap entire columns in $\mathbf{M}$
5: $idx =$ `find`(J_{qr}, i)
^ Find the index of an element with value i
6: $J_{\text{qr}}[idx] = j$

There are two important things to note about Algorithm 4. First, as seen from step 6, the pivot vector J_{qr} is updated at every iteration of the main loop. In the context of a practical algorithm, we want to preserve J_{qr} after column permutation is done, hence a copy is required. Second, the idea behind how the permutations are performed implies that the same column can be moved several times, which prevents us from parallelizing the main loop in this algorithm (hence the keyword "`sequential`" in its name).

The fact that Algorithm 4 is strictly sequential can cause performance bottlenecks on GPUs. Luckily, there is a way to introduce parallelism with some additional workspace.

REFERENCES

[1] Erik Gunnar Boman, Eric Darve, Richard B. Lehoucq, Sivasankaran Rajamanickam, Raymond S. Tuminaro, and Ichitaro Yamazaki, *Fast and robust linear solvers based on hierarchical matrices (ldrd final report)*, 2019.

[2] Peter Businger and Gene H. Golub, *Linear least squares solutions by Householder transformations*, Numerische Mathematik **7** (June 1965), no. 3, 269–276.

[3] Léopold Cambier, Chao Chen, Erik G. Boman, Sivasankaran Rajamanickam, Raymond S. Tuminaro, and Eric Darve, *An algebraic sparsified nested dissection algorithm using low-rank approximations*, SIAM Journal on Matrix Analysis and Applications **41** (January 2020), no. 2, 715–746.

[4] Coralia Cartis, Jan Fiala, and Zhen Shao, *Hashing embeddings of optimal dimension, with applications to linear least squares*, arXiv, 2021.

[5] Michael B. Cohen, *Nearly tight oblivious subspace embeddings by trace inequalities*, Proceedings of the twenty-seventh annual ACM-SIAM Symposium on Discrete Algorithms (SODA), December 2016.

[6] Michał Dereziński, Jonathan Lacotte, Mert Pilanci, and Michael W Mahoney, *Newton-LESS: Sparsification without trade-offs for the sketched Newton update*, Advances in Neural Information Processing Systems **34** (2021).

[7] P. Drineas, M. W. Mahoney, and S. Muthukrishnan, *Sampling algorithms for ℓ_2 regression and applications*, Proceedings of the 17th annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006, pp. 1127–1136.

[8] Jed A. Duersch and Ming Gu, *Randomized QR with column pivoting*, SIAM Journal on Scientific Computing **39** (January 2017), no. 4, C263–C291.

[9] Y. Gordon, *On milman's inequality and random subspaces which escape through a mesh in $\mathbb{R}^n$*, Geometric aspects of functional analysis, 1988, pp. 84–106.

[10] N. Halko, P. G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review **53** (January 2011), no. 2, 217–288.

[11] J. Kuczyński and H. Woźniakowski, *Estimating the largest eigenvalue by the power and Lanczos algorithms with a random start*, SIAM Journal on Matrix Analysis and Applications **13** (1992), no. 4, 1094–1122.

[12] P. G. Martinsson, *Blocked rank-revealing QR factorizations: How randomized sampling can be used to avoid single-vector pivoting*, 2015.

[13] Per-Gunnar Martinsson, Gregorio Quintana Ortı, Nathan Heavner, and Robert van de Geijn, *Householder QR factorization with randomization for column pivoting (HQRRP)*, SIAM Journal on Scientific Computing **39** (January 2017), no. 2, C96–C115.

[14] Riley Murray, James Demmel, Michael W. Mahoney, N. Benjamin Erichson, Maksim Melnichenko, Osman Asif Malik, Laura Grigori, Piotr Luszczek, Michał Dereziński, Miles E. Lopes, Tianyu Liang, Hengrui Luo, and Jack Dongarra, *Randomized numerical linear algebra : A perspective on the field with an eye to software*, 2023.

[15] Jelani Nelson and Huy L. Nguyen, *OSNAP: Faster numerical linear algebra algorithms via sparser subspace embeddings*, 2013 IEEE 54th annual symposium on foundations of computer science, October 2013.

[16] G. Quintana-Ortí, X. Sun, and C.H. Bischof, *A BLAS-3 version of the QR factorization with column pivoting*, SIAM Journal on Scientific Computing **19** (1998), no. 5, 1486–1494. Posted in preprint form as LAWN 114 in 1996.

[17] V Rokhlin and M Tygert, *A fast randomized algorithm for overdetermined linear least-squares regression*, Proceedings of the National Academy of Sciences **105** (September 2008), no. 36, 13212–13217.

[18] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw, *Parallel random numbers: As easy as 1, 2, 3*, Sc '11: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis, 2011, pp. 1–12.

[19] Tamas Sarlos, *Improved approximation algorithms for large matrices via random projections*, Proceedings of the 47th annual IEEE Symposium on Foundations of Computer Science (FOCS), 2006, pp. 143–152.

[20] Sergey Voronin and Per-Gunnar Martinsson, *Efficient algorithms for CUR and interpolative matrix decompositions*, Advances in Computational Mathematics **43** (November 2016), no. 3, 495–516.

[21] Franco Woolfe, Edo Liberty, Vladimir Rokhlin, and Mark Tygert, *A fast randomized algorithm for the approximation of matrices*, Applied and Computational Harmonic Analysis **25** (2008), no. 3, 335–366.

[22] Jianwei Xiao, Ming Gu, and Julien Langou, *Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations*, 2017 IEEE 24th international conference on high performance computing (HiPC), 2017, pp. 233–242.

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address
1	Riley Murray	Riley Murray 230 Upland Road Unit 2 Cambridge, MA 02140

This page intentionally left blank.

This page intentionally left blank.

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.