LOW-RANK APPROXIMATION OF PARAMETER-DEPENDENT MATRICES VIA CUR DECOMPOSITION *

TAEJUN PARK† AND YUJI NAKATSUKASA†

Abstract. A low-rank approximation of a parameter-dependent matrix $A(t)$ is an important task in the computational sciences appearing for example in dynamical systems and compression of a series of images. In this work, we introduce AdaCUR, an efficient algorithm for computing a low-rank approximation of parameter-dependent matrices via CUR decompositions. The key idea for this algorithm is that for nearby parameter values, the column and row indices for the CUR decomposition can often be reused. AdaCUR is rank-adaptive, provides error control, and has complexity that compares favorably against existing methods. A faster algorithm which we call FastAdaCUR that prioritizes speed over accuracy is also given, which is rank-adaptive and has complexity which is at most linear in the number of rows or columns, but without error control.

Key words. Parameter-dependent matrices, low-rank approximation, CUR decomposition, randomized algorithms

MSC codes. 15A23, 65F55, 68W20

1. Introduction.

The task of finding a low-rank approximation to a matrix is ubiquitous in the computational sciences [56]. In large scale problems, low-rank approximation provides an efficient way to store and process the matrix. In this work, we study the low-rank approximation of a parameter-dependent matrix $A(t) \in \mathbb{R}^{m\times n}$ with $m \geq n$ for a finite number of parameter values $t \in \mathbb{R}$ in some compact domain $D \subset \mathbb{R}$. This task has appeared in several applications, for example, in compression of a series of images [48], dynamical systems [36] and Gaussian random fields [38].

When $A(t) \equiv A$ is a constant, the truncated singular value decomposition (TSVD) attains the best low-rank approximation in any unitarily invariant norm for a fixed target rank r [34, § 7.4.9]. However, the cost of computing the truncated SVD becomes infeasible for large matrices. The situation is exacerbated when $A(t)$ varies with parameter t, requiring the evaluation of the truncated SVD at every parameter value t of interest. Consequently, numerous alternative approaches have been explored in recent years such as dynamical low-rank approximation [36, 48], randomized SVD and generalized Nyström method [37] and the CUR decomposition [17].

In this work, we use the CUR decomposition [25, 42] to find a low-rank approximation to a parameter-dependent matrix $A(t)$. The CUR decomposition of a matrix A is a low-rank approximation that takes the form

$$A \approx A(:, J)A(I, J)^\dagger A(I, :) = CU^\dagger R =: A_{IJ} \tag{1.1}$$

in MATLAB notation. Here, $C = A(:, J)$ is a subset of the columns of A with J being the column indices, $R = A(I, :)$ is a subset of the rows of A with I being the row indices and $U = A(I, J)$ is the intersection of C and R.[1] While the CUR decomposition may be considered suboptimal in comparison to the truncated SVD,

*Date: February 26, 2025

Funding: TP is supported by the Heilbronn Institute for Mathematical Research. YN is supported by EPSRC grants EP/Y010086/1 and EP/Y030990/1.

†Mathematical Institute, University of Oxford, Oxford, OX2 6GG, UK, (park@maths.ox.ac.uk, nakatsukasa@maths.ox.ac.uk).

[1]There are other choices for the CUR decomposition. In particular, the choice $A \approx CC^\dagger AR^\dagger R$ minimizes the Frobenius norm error given C and R. We choose $A = CU^\dagger R$ in this paper as it is faster to compute.

it possesses many favorable properties. Firstly, given the row and column indices, they are extremely efficient to compute and store as they do not require reading the entire matrix A. Additionally, as the low-rank factors are the subsets of the original matrix A, they inherit certain properties of A such as sparsity and non-negativity. Moreover, these factors assist with data interpretation by revealing the important rows and columns. For further insight and theory on the CUR decomposition, see, for example, [6, 29, 42].

For any set of row indices I with $|I| = r$ and column indices J with $|J| = r$ where r is the target rank, the CUR decomposition satisfies the following error bound [50]

$$\left\|A - CU^{\dagger}R\right\|_F \leq \left\|Q_C(I,:)^{\dagger}\right\|_2 \left\|Q_X(J,:)^{-1}\right\|_2 \left\|A - AX^{\dagger}X\right\|_F \tag{1.2}$$

where $X \in \mathbb{R}^{r\times n}$ is any row space approximator of A and $Q_X \in \mathbb{R}^{n\times r}$ and $Q_C \in \mathbb{R}^{m\times r}$ are orthonormal matrices spanning the columns of X^T and $C = A(:,J)$ respectively.[2] The first two terms on the right-hand side of the inequality $\left\|Q_C(I,:)^{\dagger}\right\|_2$ and $\left\|Q_X(J,:)^{-1}\right\|_2$ mostly govern the accuracy of the CUR decomposition, because the row space approximator X is arbitrary and can be chosen such that $\left\|A - AX^{\dagger}X\right\|_F$ is quasi-optimal. Therefore, it is important to get a good set of row and column indices that control the first two terms. There are many existing algorithms that lead to a good set of indices such as leverage score sampling [20, 42], DPP sampling [15], volume sampling [14, 16] and pivoting strategies [19, 21, 24, 27, 52, 53, 57], which all attempt to minimize the first two terms. Notably, there exists a set of row and column indices for which the CUR decomposition satisfies the following near-optimal guarantee [61]

$$\left\|A - CU^{\dagger}R\right\|_F \leq (1+r)\left\|A - [\![A]\!]_r\right\|_F$$

where $[\![A]\!]_r$ is the best rank-r approximation to A. This means that little is lost by requiring our low-rank approximation to be in CUR form, as long as the indices are chosen appropriately. Polynomial-time algorithms for constructing such indices can be found in [14, 49]. The bound (1.2) can be improved by oversampling the rows, that is, by obtaining more row indices such that I has more indices than the target rank. The topic of oversampling has been explored in [1, 23, 50, 51, 62]. Notably, [50] describes how to compute the CUR decomposition in a numerically stable way and shows that oversampling can help improve the stability (in addition to accuracy) of the CUR decomposition.

Now, having defined what the CUR decomposition is, the objective of this work is the following:

Objective 1.1. *Let $A(t)$ be a parameter-dependent matrix. Then given a set of parameter values $t_1, t_2, ..., t_q$, and a tolerance ϵ, devise an algorithm that approximately achieves*

$$\left\|A(t_i) - \left(A(t_i)(:,J_i)\right)\left(A(t_i)(I_i,J_i)^{\dagger}\right)\left(A(t_i)(I_i,:)\right)\right\|_F \leq \epsilon\left\|A(t_i)\right\|_F \tag{1.3}$$

for each i, where I_i and J_i are the row and column indices corresponding to t_i.

For parameter-dependent matrices $A(t)$, we can naively recompute the row and the column indices for each parameter value of interest. However, this approach

[2] This bound can also be stated in terms of a column approximator and a subset of the rows of A.

can be inefficient and wasteful, as the computed indices for one parameter value are likely to contain information about nearby parameter values. The reason is because a matrix may only undergo slight deviations when the parameter value changes. More concretely, let $A(t)$ be a parameter-dependent matrix and I and J be a set of row and column indices respectively. Then by setting $X = V_r(t)^T$ where $V_r(t)$ contains the r-dominant right singular vectors of $A(t)$, (1.2) changes to

(1.4)
$$\left\|A(t) - C(t)U(t)^\dagger R(t)\right\|_F \leq \left\|Q_{C(t)}(I,:)^\dagger\right\|_2 \left\|Q_{V_r(t)}(J,:)^{-1}\right\|_2 \|A(t) - [\![A(t)]\!]_r\|_F$$

where $[\![A(t)]\!]_r$ is the best rank-r approximation to $A(t)$. Now the rightmost term in (1.4) is optimal for any parameter value t, so we compare the first two terms. Suppose we have two different parameter values, say t_1 and t_2. Then, if both $\left\|Q_{C(t_1)}(I,:)^\dagger\right\|_2 \left\|Q_{V_r(t_1)}(J,:)^{-1}\right\|_2$ and $\left\|Q_{C(t_2)}(I,:)^\dagger\right\|_2 \left\|Q_{V_r(t_2)}(J,:)^{-1}\right\|_2$ remain small with the same set of indices I and J, we can use I and J to approximate both $A(t_1)$ and $A(t_2)$.[3]

More specifically, define the following set of parameter-dependent matrices:

(1.5)
$$\mathcal{M}_{IJ}^{(r)}(\delta) := \left\{A(t) \in \mathbb{R}^{m\times n} : \left\|Q_{A(t)(:,J)}(I,:)^\dagger\right\|_2 \left\|Q_{V(t)}(J,:)^{-1}\right\|_2 \leq \delta \text{ for all } t \in D\right\}$$

for some domain $D \subset \mathbb{R}$ and sets of indices I and J with $|I| = |J| = r$. Then, for all $A(t) \in \mathcal{M}_{IJ}^{(r)}(\delta)$,

$$\left\|A(t) - C(t)U(t)^\dagger R(t)\right\|_F \leq \delta \|A(t) - [\![A(t)]\!]_r\|_F \tag{1.6}$$

holds for all $t \in D$. Therefore if $A(t) \in \mathcal{M}_{IJ}^{(r)}(\delta)$ for sufficiently small $\delta \geq 1$ then the index sets I and J provide a good rank-r CUR approximation to $A(t)$ for all $t \in D$. An example of a class of parameter-dependent matrices belonging to $\mathcal{M}_{IJ}^{(r)}(\delta)$ are incoherent rank-r parameter-dependent matrices. In general, it is difficult to precisely determine whether a parameter-dependent matrix belongs to $\mathcal{M}_{IJ}^{(r)}(\delta)$ for some (practical) starting indices I and J or not, but intuitively, any parameter-dependent matrix that is sufficiently low-rank and has singular vectors that change gradually should belong to $\mathcal{M}_{IJ}^{(r)}(\delta)$ for some I, J, δ and r. We do not explore the technicalities of this further in this work.

Nevertheless, in practice, for nearby parameter values, the identical set of indices I and J often capture the important rows and columns of both $A(t_1)$ and $A(t_2)$; see Sections 3 and 4. This motivates us to reuse the indices for various parameter values, potentially resulting in significant savings, often up to a factor r in complexity where r is the target rank. With this key idea in mind, the main goal of this paper is to devise a rank-adaptive certified[4] algorithm that reuses the indices as much as possible until the error becomes too large, at which point we recompute the indices.

To achieve this goal efficiently and reliably, we rely on a variety of efficient tools, many of which are borrowed from the randomized linear algebra literature such as randomized rank estimation [2, 45] and pivoting on a random sketch [19, 22, 57]; see Section 2. We assume throughout that for each parameter t, $A(t)$ has decaying singular values so that a low-rank approximation is possible. This is clearly a required assumption for low-rank approximation techniques to be effective.

[3] In our work, we allow $\|A(t_1) - A(t_2)\|_2$ to be large, say $\mathcal{O}(1)$, and therefore the perturbation bounds for the CUR decomposition in [30] is not enough to explain why the same indices can be reused for nearby parameter values.

[4] We use the term *certified* to denote approximation methods that are designed to control the approximation error.

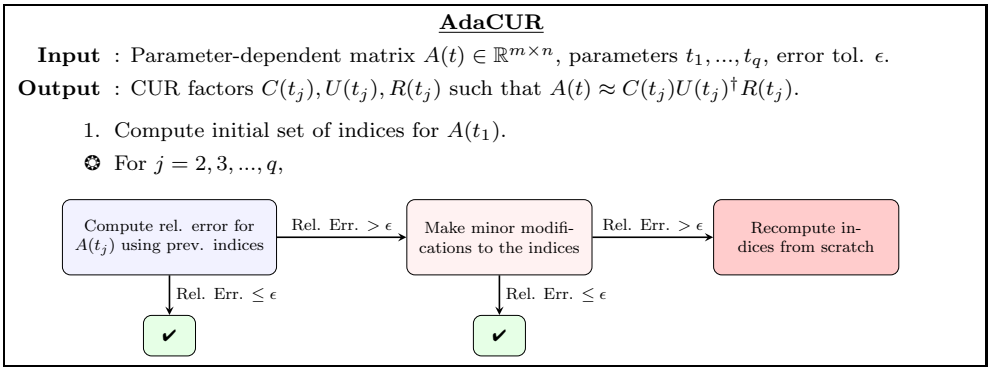


FIG. 1. *An overview of AdaCUR*

In the next section, we provide brief outlines of our algorithms, AdaCUR and FastAdaCUR, review existing methods for approximating parameter-dependent matrices, and summarize our contributions. This is followed by an overview of the various techniques from (randomized) numerical linear algebra that we use in our algorithms. The next two sections cover AdaCUR and FastAdaCUR in detail, as well as numerical experiments highlighting their performance. Finally, we conclude and discuss topics for further research.

1.1. Brief outline of AdaCUR and FastAdaCUR. Before introducing the two algorithms, AdaCUR and FastAdaCUR, in detail in Section 3, we first provide a brief outline of their core ideas.

AdaCUR is aimed at efficiently computing an accurate low-rank approximation of parameter-dependent matrices. A high-level overview of AdaCUR is given in Figure 1.

AdaCUR begins by computing an initial set of indices for $A(t_1)$. For subsequent parameter values t_j, AdaCUR first computes the relative error for $A(t_j)$ using the previous sets of indices. If the error is smaller than the tolerance ϵ, the algorithm retains the previous sets of indices and proceeds to the next parameter value. If the error exceeds the tolerance, minor modifications are made to the indices, and the algorithm checks whether the tolerance is satisfied. If the modified indices meet the tolerance, they are used; otherwise, the indices are recomputed from scratch. The precise details of AdaCUR are outlined in Section 3.2.

FastAdaCUR is aimed at achieving even greater speed at the cost of possible reduction in accuracy. Instead of computing the error which can be expensive, FastAdaCUR keeps a small set of extra indices that helps with rank adaptivity. A high-level overview of FastAdaCUR is given in Figure 2.

FastAdaCUR begins by computing an initial sets of indices I, J and a small set of additional indices I_s, J_s for $A(t_1)$. For subsequent parameter values t_j, FastAdaCUR calculates the ϵ-rank of the core matrix $A(I \cup I_s, J \cup J_s)$ using the previous sets of indices to determine whether the ϵ-rank exceeds the size of the index sets I and J. Importantly, for FastAdaCUR, the entire matrix is not accessed beyond the first parameter value, which makes the algorithm efficient. If the ϵ-rank has increased, indices are added to I and J from I_s and J_s. Conversely, if the ϵ-rank has decreased, indices are removed from I and J. After this adjustment, the algorithm proceeds to the next parameter value. The details of FastAdaCUR are provided in Section 3.3. Unlike AdaCUR, FastAdaCUR does not have error control mechanism, making

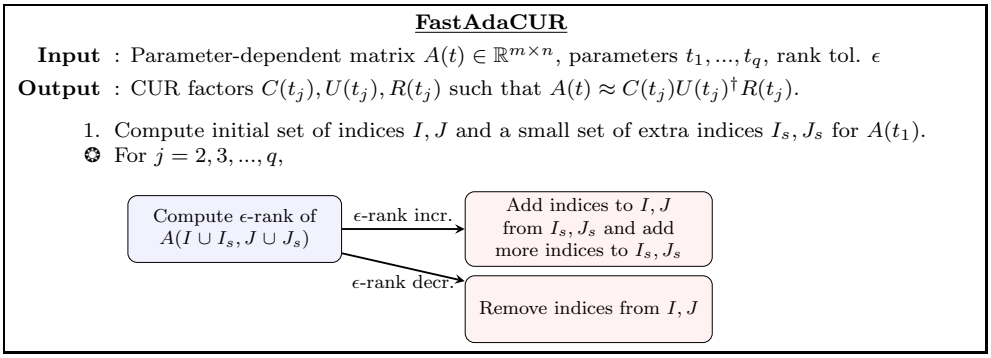


FIG. 2. *An overview of FastAdaCUR*

it vulnerable to adversarial examples; see Section 4.4.

Existing methods. There have been several different approaches for finding a low-rank approximation to parameter-dependent matrices $A(t)$. We describe four different classes of methods. First, dynamical low-rank approximation is a differential-equation-based approach where the low-rank factors are obtained from projecting the time derivative of $A(t)$, $\dot{A}(t)$, onto the tangent space of the smooth manifold consisting of fixed-rank matrices; see [40]. A number of variations from the original dynamical low-rank approximation [36] have been proposed to deal with various issues such as the stiffness introduced by the curvature of the manifold [9, 35, 41] and rank-adaptivity [8, 31, 32, 33]. The complexity of this approach is typically $\mathcal{O}(r_i T_{A(t_i)})$ for the parameter t_i, where r_i is the target rank for $A(t_i)$ and $T_{A(t_i)}$ is the cost of matrix-vector product with $A(t_i)$ or $A(t_i)^T$. Secondly, Kressner and Lam [37] use randomized SVD [28] and generalized Nyström [46, 55], both based on random projections, to find low-rank approximations of parameter-dependent matrices $A(t)$. They focus on matrices that admit an affine linear decomposition with respect to t. Notably, they use the same random embedding for all parameter values rather than generating a new random embedding for each parameter value. The complexity is also $\mathcal{O}(r_i T_{A(t_i)})$ for the parameter value t_i. Next, Donello et al. [17] use the CUR decomposition to find a low-rank approximation to parameter-dependent matrices. At each iteration, a new set of column and row indices for the CUR decomposition are computed by applying the discrete empirical interpolation method (DEIM) [10, 21, 52] to the singular vectors of the CUR decomposition from the previous iteration. While this approach benefits from not viewing the entire matrix, it may suffer from the same adversarial example as the fast algorithm, FastAdaCUR discussed in this paper; see Algorithm 3.3 and Section 4.4. They allow oversampling of column and/or row indices to improve the accuracy of the CUR decomposition and propose rank-adaptivity where the rank is either increased or decreased by 1 if the smallest singular value of the current iterate is larger or smaller, respectively, than some threshold. Although their algorithm is based on the CUR decomposition, the low-rank factors in their approach are represented in the form of an SVD. This SVD is computed from the CUR decomposition at each iteration, given that orthonormal factors are necessary in DEIM. The complexity for this algorithm is $\mathcal{O}((m+n)r_i^2)$ for the parameter value t_i. Lastly, in the special case when $A(t)$ is symmetric positive definite, a parametric variant of adaptive cross approximation has been used in [38].

Contributions. The contribution of this paper lies in the design of efficient rank-adaptive algorithms for low-rank approximations of parameter-dependent matrices. The algorithms are based on the CUR decomposition of a matrix. We first demonstrate that the same set of row and column indices, or making slight modifications to the indices, can yield effective low-rank approximations for nearby parameter values. This observation forms the basis of our efficient algorithms, which we describe in Section 3. We present two distinct algorithms: a rank-adaptive algorithm with error control, which we call AdaCUR (Adaptive CUR), and a faster rank-adaptive algorithm that sacrifices error control for speed, which we name FastAdaCUR (Fast Adaptive CUR); see Algorithms 3.2 and 3.3 respectively.

AdaCUR has the worst-case time complexity of $\mathcal{O}\left(r_i T_{A(t_i)} + (m+n)r_i^2\right)$ for the parameter value t_i where r_i is the target rank for $A(t_i)$ and $T_{A(t_i)}$ denotes the cost of matrix-vector product with $A(t_i)$ or $A(t_i)^T$. However, in practice, the algorithm frequently runs with the best-case time complexity of $\mathcal{O}(T_{A(t_i)} + (m+n)r_i)$ for the parameter value t_i; see Sections 3.2 and 4. This is competitive with existing methods such as [9, 37, 41], which run with complexity $\mathcal{O}(r_i T_{A(t_i)})$. Notably, our algorithm's rank-adaptive nature and built-in error control offer distinct advantages over many existing algorithms, which often lack one or both of these features.

FastAdaCUR, aside from the initial phase of computing the initial indices, runs linearly in m and n in the worse-case. The best-case complexity is $\mathcal{O}(r_i^3)$ for the parameter value t_i, which makes the algorithm remarkably fast. While this algorithm is also rank-adaptive, it lacks rigorous error control since its priority is efficiency over accuracy. However, the algorithm has the advantage of needing only $\mathcal{O}(r)$ rows and columns for each parameter value; i.e., there is no need to view the full matrix. This feature is particularly attractive when the entries are expensive to compute. In experiments, we notice that the error may grow as we iterate through the parameter for difficult problems. Here, difficult problems refer to problems where $A(t)$ undergoes frequent large rank changes or one that changes rapidly such that the previous indices carry little to no information for the next. Nevertheless, we frequently observe that the algorithm performs well on easier problems; see Section 4.

Notation. Throughout, we use $\|\cdot\|_2$ for the spectral norm or the vector-ℓ_2 norm and $\|\cdot\|_F$ for the Frobenius norm. We use dagger † to denote the pseudoinverse of a matrix and $[\![A]\!]_r$ to denote the best rank-r approximation to A in any unitarily invariant norm, i.e., the approximation derived from truncated SVD [34, § 7.4.9]. We use $a_1 \lesssim a_2$ to denote that $a_1 \leq Ca_2$ for some constant $C > 0$. Unless specified otherwise, $\sigma_i(A)$ denotes the ith largest singular value of the matrix A. We use MATLAB style notation for matrices and vectors. For example, for the kth to $(k+j)$th columns of a matrix A we write $A(:, k:k+j)$. Lastly, we use $|I|$ to denote the length of the vector or the cardinality of the set I, $I_1 - I_2$ to be the set difference between I_1 and I_2, and define $[n] := \{1, 2, ..., n\}$.

2. Preliminaries. In this section, we discuss the tools needed for our proposed methods introduced in Section 3. Many of the methods described here are in the randomized numerical linear algebra literature. For an in-depth review, we refer to [28, 44, 58]. First, in Section 2.1, we discuss the core ingredient in many randomized numerical linear algebra algorithms, which are random embeddings. Following this, in Section 2.2, we review pivoting on a random sketch, which efficiently computes the indices for the CUR decomposition, and in Section 2.3, we review a fast randomized algorithm for computing the numerical rank of a matrix. Lastly, in Section 2.4, we discuss an efficient method for computing the Frobenius norm of a matrix, which is

used for error estimation for AdaCUR in Section 3.

2.1. Random embeddings. Let $A \in \mathbb{R}^{m\times n}$ be a matrix of rank $r \leq \min\{m,n\}$ (typically $r \ll \min\{m,n\}$). Then $\Gamma \in \mathbb{R}^{s\times m}$ ($r \leq s \leq \min\{m,n\}$) is a subspace embedding for the span of A with distortion $\epsilon \in (0,1)$ if

$$(1-\epsilon)\,\|Ax\|_2 \leq \|\Gamma Ax\|_2 \leq (1+\epsilon)\,\|Ax\|_2 \tag{2.1}$$

for every $x \in \mathbb{R}^n$. Therefore, Γ is a linear map which preserves the 2-norm of every vector in a given subspace. A random embedding is a subspace embedding drawn at random that satisfies (2.1) for all A with high probability. A typical application of random embeddings is in *matrix sketching*, a technique for dimensionality reduction. Matrix sketching compresses the original matrix into a smaller-sized matrix while retaining much of its original information. For example, for a matrix $A \in \mathbb{R}^{m\times n}$ of rank $r \ll \min\{m,n\}$, $\Gamma A \in \mathbb{R}^{s\times n}$ where $r \leq s \ll \min\{m,n\}$ is a (row) sketch of the matrix A. Here, the integer s is called the sketch size.

There are a few important examples of random embeddings such as Gaussian embeddings [28, 44], subsampled randomized trigonometric transforms (SRTTs) [5, 54] and sparse embeddings [12, 13, 47]. We focus on Gaussian embeddings in this work. A Gaussian embedding $\Gamma \in \mathbb{R}^{s\times m}$ has i.i.d. entries $\Gamma_{ij} \sim \mathcal{N}(0, 1/s)$. Here, $s = \Omega(r/\epsilon^2)$ for theoretical guarantees, but $s = r + \Omega(1)$ works well in practice. The cost of applying a Gaussian embedding to a matrix $A \in \mathbb{R}^{m\times n}$ is $\mathcal{O}(sT_A)$ where T_A is the cost of matrix-vector multiply with A or A^T. Other random embeddings provide a similar theoretical guarantee but generally require a larger sketch size, say $s = \Omega(r \log r/\epsilon^2)$. However, they are usually cheaper to apply; for example, SRTTs cost $\mathcal{O}(mn \log s)$ [59].

2.2. Pivoting on a random sketch. Pivoting on a random sketch [19, 22, 57] is an attractive and efficient method for finding a spanning set of columns and/or rows of a matrix. This will be the core part for computing the CUR decomposition. This method has two main steps: sketch and pivot. Let $A \in \mathbb{R}^{m\times n}$ be a matrix and r be the target rank. Then the basic idea works as follows for column indices:

1. *Sketch*: Draw a random embedding $\Gamma \in \mathbb{R}^{r\times m}$ and form $X = \Gamma A \in \mathbb{R}^{r\times n}$.[5]
2. *Pivot*: Perform a pivoting scheme, e.g., column pivoted QR (CPQR) or partially pivoted LU (LUPP) on X. Collect the chosen pivot indices.

The sketching step entails compressing the original matrix down to a smaller-sized matrix using a random embedding. The sketch $X = \Gamma A$ forms a good approximation to the row space of A [28]. The pivoting step involves applying a pivoting scheme on the smaller-sized matrix X, which reduces the cost compared to applying the pivoting scheme directly on A. There are many pivoting schemes such as column pivoted QR (CPQR), partially pivoted LU (LUPP) or those with strong theoretical guarantees such as Gu-Eisenstat's strong rank-revealing QR (sRRQR) [27] or BSS sampling [3, 4]. See [19] for a comparison. In this work, we use a version of Algorithm 1 from [19], which we state below in Algorithm 2.1.

Algorithm 2.1 selects the column indices first by applying CPQR on the sketch $X = \Gamma A$, and then selects the row indices by applying CPQR on the chosen columns $A(:, J)^T$.[6] The complexity of Algorithm 2.1 is $\mathcal{O}(rT_A + (m+n)r^2)$, where rT_A is the

[5] For robustness, oversampling is recommended, that is, we draw a random embedding with the sketch size larger than r, say $2r$, in step 1 and obtain r pivots in step 2. See [18, 19] for a discussion.

[6] The fourth step in Algorithm 2.1, which involves applying CPQR on the selected columns instead of, for example, a column sketch of A, can be important for the stability and accuracy of the CUR decomposition [50].

Algorithm 2.1 Pivoting on a random sketch

Input: $A \in \mathbb{R}^{m\times n}$, target rank r (typically $r \ll \min\{m,n\}$)
Output: Column indices J and row indices I with $|I| = |J| = r$

`function` $[I, J] =$ `Rand_Pivot`(A, r)
1: Draw a Gaussian embedding $\Gamma \in \mathbb{R}^{r\times m}$.
2: Set $X = \Gamma A \in \mathbb{R}^{r\times n}$, a row sketch of A
3: Apply CPQR on X. Let J be the r column pivots.
4: Apply CPQR on $A(:, J)^T$. Let I be the r row pivots.

cost of forming the Gaussian row sketch X in line 2 (see Section 2.1) and $\mathcal{O}(nr^2)$ and $\mathcal{O}(mr^2)$ are the cost of applying CPQR on X (line 3) and $A(:, J)^T$ (line 4) respectively.

2.3. Randomized rank estimation. Randomized rank estimation [2, 45] is an efficient tool for finding the ϵ-rank of a matrix $A \in \mathbb{R}^{m\times n}$. As we often require the target rank as input, for example in Algorithm 2.1, we would like an efficient method for computing the ϵ-rank of a matrix. This method also relies on sketching but, unlike pivoting on a random sketch, it involves sketching Γ_1, Γ_2 from both sides of the matrix. The idea is to approximate the ϵ-rank of A using

$$\hat{r}_\epsilon = \mathrm{rank}_\epsilon(\Gamma_1 A \Gamma_2) \tag{2.2}$$

where $\mathrm{rank}_\epsilon(B)$ is the ϵ-rank of the matrix B (i.e., the number of singular values larger than ϵ) and $\Gamma_1 A\Gamma_2 \in \mathbb{R}^{s\times 2s}$ with $s \gtrsim \mathrm{rank}_\epsilon(A)$. Here, $s \gtrsim \mathrm{rank}_\epsilon(A)$ is achieved by gradually increasing s by appending to the sketches until $\sigma_{\min}(\Gamma_1 A\Gamma_2) < \epsilon$; see [45] for details. The overall cost of the algorithm is $\mathcal{O}(sT_A + ns\log s + s^3)$ where $\mathcal{O}(sT_A)$ is the cost of forming the Gaussian sketch, $\mathcal{O}(ns\log s)$ is the cost of forming the SRTT sketch and $\mathcal{O}(s^3)$ is the cost of computing the singular values of $\Gamma_1 A\Gamma_2$.

Combining pivoting on a random sketch and randomized rank estimation. Since we require the target rank as input for pivoting on a random sketch (Algorithm 2.1), randomized rank estimation and pivoting on a random sketch can be used together. Both algorithms need to form the row sketch $X = \Gamma A$ as part of their algorithm. Therefore, we can input the row sketch X obtained from randomized rank estimation into pivoting on a random sketch. This avoids the need to reform the row sketch making randomized rank estimation almost free when used with pivoting on a random sketch. The overall algorithm is outlined in Algorithm 2.2, giving the overall complexity of $\mathcal{O}(\hat{r}_\epsilon T_A + (m+n)\hat{r}_\epsilon^2)$. By combining the two algorithms into one, the number of matrix-vector products with A is halved, which is often the dominant cost.

Algorithm 2.2 Pivoting on a random sketch using randomized rank estimation

Input: $A \in \mathbb{R}^{m\times n}$, tolerance ϵ
Output: Row indices I and Column indices J

`function` $[I, J] =$ `Rand_Pivot_RankEst`(A, ϵ)
1: $[\hat{r}_\epsilon, X] =$ Estimate the ϵ-rank rank using (2.2) ▷ $X = \Gamma_1 A$ is the row sketch from (2.2)
2: $[I, J] =$ `Rand_Pivot`$(A, X, \hat{r}_\epsilon)$ ▷ Take the row sketch X as input in Algorithm 2.1

2.4. Randomized norm estimation. In order to monitor how well our algorithm performs, we need an efficient way to estimate the norm of the error. Randomized norm estimation [26, 44] is an efficient method for finding an approximation to a norm of a matrix. In this work, we focus on the Frobenius norm as it is a natural choice for low-rank approximation and easier to estimate. Let $A \in \mathbb{R}^{m\times n}$ be a matrix

and $\hat{A}_r \in \mathbb{R}^{m\times n}$ be a rank-r approximation to A. Then we would like to estimate $\left\|A - \hat{A}_r\right\|_F$. As before, we can sketch to get an approximation to $A - \hat{A}_r$ and then compute the Frobenius norm. It turns out that the sketch size of $O(1)$, say 5, suffices. More specifically, we have the following theorem from [26].

THEOREM 2.1. *[26, Theorem 3.1] Let $A \in \mathbb{R}^{m\times n}$ be a matrix, $\Gamma \in \mathbb{R}^{s\times m}$ be a Gaussian matrix with i.i.d. entries $\Gamma_{ij} \sim \mathcal{N}(0,1)$ and set $\rho := \|A\|_F^2 / \|A\|_2^2$. For any $\tau > 1$ and $s \le m$,*

(2.3)
$$\Pr\left(\frac{\|A\|_F}{\tau} < \frac{\|\Gamma A\|_F}{\sqrt{s}} \le \tau \|A\|_F\right) \ge 1 - \exp\left(-\frac{s\rho}{2}(\tau-1)^2\right) - \exp\left(-\frac{s\rho}{4}\frac{(\tau^2-1)^2}{\tau^4}\right)$$

Setting $\|A\|_F = 5\|A\|_2$, $\tau = 2$ and $s = 5$, the above theorem tells us

$$\Pr\left(\frac{\|A\|_F}{2} < \frac{\|\Gamma A\|_F}{\sqrt{5}} \le 2\|A\|_F\right) \ge 1 - 2.33\cdot 10^{-8}, \tag{2.4}$$

i.e. with a small number s of Gaussian vectors, we can well-approximate the Frobenius norm of a matrix. The algorithm specialized to CUR decomposition is presented in Algorithm 2.3.

Algorithm 2.3 Randomized norm estimation for CUR decomposition

Input: $A \in \mathbb{R}^{m\times n}$, I row indices, J column indices, sample size s (rec. $s = 5$)
Output: E, an estimate for $\left\|A - A(:,J)A(I,J)^\dagger A(I,:)\right\|_F$

`function` $E =$ `Norm_Est`(A, I, J, s)
1: Draw a Gaussian embedding $\Gamma \in \mathbb{R}^{s\times m}$
2: Set $X = \Gamma A \in \mathbb{R}^{s\times n}$
3: Set $\hat{X} = \Gamma A(:,J)A(I,J)^\dagger A(I,:) \in \mathbb{R}^{s\times n}$
4: $E = \left\|X - \hat{X}\right\|_F$

The complexity of Algorithm 2.3 is $\mathcal{O}(sT_A + (m+n)rs) = \mathcal{O}(T_A + (m+n)r)$ where $r = \max\{|I|, |J|\}$ since $s = \mathcal{O}(1)$. This method is particularly useful when A can only be accessed through matrix-vector multiply.

In addition to approximating the error using randomized norm estimation, we obtain, as a by-product, a small sketch of the low-rank residual matrix since

$$X - \hat{X} = \Gamma(A - A(:,J)A(I,J)^\dagger A(I,:)). \tag{2.5}$$

This small sketch can be used to obtain an additional set of row and column indices by using pivoting on it, similarly to Algorithm 2.1. Specifically, we apply pivoting on the sketched residual $X - \hat{X}$ to get an extra set of s column indices, denoted J_s. Next, we apply pivoting on the chosen columns of the residual matrix,

$$A(:,J_s) - A(:,J)A(I,J)^\dagger A(I,J_s)$$

to extract an extra set of s row indices I_s. It is important to note that I_s and J_s will not include indices already chosen in I and J because the residual matrix $A - A(:,J)A(I,J)^\dagger A(I,:)$ is zero in the rows and columns corresponding to I and J. The procedure for obtaining s additional indices from the sketched residual is outlined in Algorithm 2.4.

The complexity of Algorithm 2.4 is $\mathcal{O}((m+n)s^2 + mrs + r^3) = \mathcal{O}(n + mr + r^3)$, where $r = \max\{|I|, |J|\}$ since $s = \mathcal{O}(1)$. The dominant cost lies in the computation

Algorithm 2.4 Pivoting on the residual matrix

Input: $X_R \in \mathbb{R}^{s\times n}$ sketch of the residual matrix, I row indices, J column indices
Output: I_s and J_s, extra sets of indices

`function` $[I_s, J_s] =$ `Pivot_Residual`(X_R, I, J)
1: Apply CPQR on X_R. Let J_s be the s column pivots,
2: Set $C_R = A(:, J_s) - A(:, J)A(I, J)^\dagger A(I, J_s) \in \mathbb{R}^{m\times s}$
3: Apply CPQR on C_R^T. Let I_s be the s row pivots.

of $A(:, J)A(I, J)^\dagger A(I, J_s)$, which requires $\mathcal{O}(mr + r^3)$ operations. By adding these additional indices, we can improve the accuracy of the CUR decomposition. Algorithm 2.4 will be applied later in AdaCUR to refine the approximation.

Multiple error estimation. Algorithm 2.3 can easily be used to estimate multiple low-rank approximations of A. Suppose we are given two sets of indices I_1 and J_1, and I_2 and J_2. Then once the row sketch of A, $X = \Gamma A$ has been computed, we only need to compute the row sketches of the low-rank approximations $\hat{X}_1 = \Gamma A_{I_1 J_1}$ and $\hat{X}_2 = \Gamma A_{I_2 J_2}$ to approximate the error. Here, $A_{IJ} = A(:, J)A(I, J)^\dagger A(I, :)$. Since forming X is typically the dominant cost in Algorithm 2.3, this allows us to efficiently test multiple low-rank approximations.

3. Proposed method. Let us first restate the problem. Let $A(t) \in \mathbb{R}^{m\times n}$ be a parameter-dependent matrix with $t \in D$ where $D \subseteq \mathbb{R}$ is a compact domain and $t_1, t_2, ..., t_q \in D$ be a finite number of distinct sample points ordered in some way, e.g., $t_1 < t_2 < \cdots < t_q$ for $t \in \mathbb{R}$. Then find a low-rank approximation of $A(t_1), A(t_2), ..., A(t_q)$.[7]

The goal of this work is to devise an efficient algorithm that reuses the indices to their fullest extent in the CUR decomposition as we iterate through the parameter values. To see how we can use the same indices for different parameter values, recall (1.2) with oversampling from [50]:

$$\|A - A_{I\cup I_0, J}\|_F \le \left\|Q_C(I \cup I_0, :)^\dagger\right\|_2 \left\|V_r(J, :)^{-1}\right\|_2 \|A - [\![A]\!]_r\|_F . \tag{3.1}$$

Here, I_0 is an extra set of row indices resulting from oversampling that are distinct from I.

The bound (3.1) leads us to two observations. First, as described earlier in the introduction, if $A(t) \in \mathcal{M}_{IJ}^{(r)}(\delta)$, then

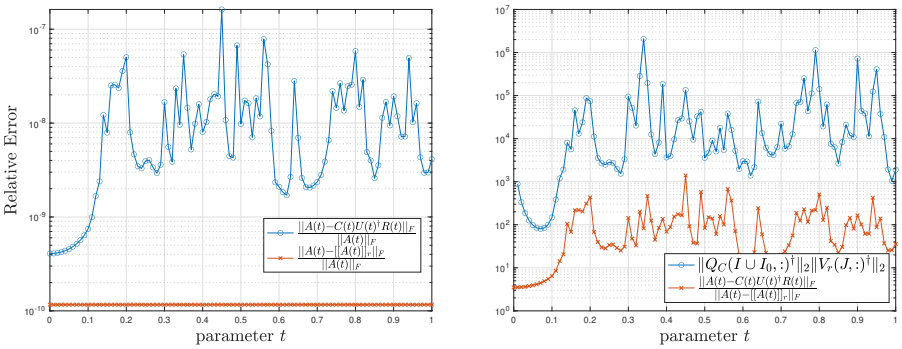
$$\|A - A_{I\cup I_0, J}\|_F \le \delta \|A - [\![A]\!]_r\|_F .$$

Therefore, if δ is sufficiently small, say $\mathcal{O}(10)$, the sets of indices $I \cup I_0$ and J yield a good CUR approximation. However, this does not imply that a high value of δ necessarily results in a poor CUR approximation. Let us illustrate this with a numerical example. We use the synthetic example from [9] given by

$$A(t) = e^{tW_1} e^t D e^{tW_2}, t \in [0, 1] \tag{3.2}$$

where $D \in \mathbb{R}^{n\times n}$ is diagonal with entries $d_{jj} = 2^{-j}$ and $W_1, W_2 \in \mathbb{R}^{n\times n}$ are randomly generated skew-symmetric matrices. The singular values of $A(t)$ are $e^t 2^{-j}$ for $j = 1, 2, ..., n$. We take $n = 500$ and take 101 equispaced points in the interval $[0, 1]$ for t. The matrix exponentials were computed using the `expm` command in MATLAB.

[7] Although our algorithms are presented for a finite number of sample points, they are likely to work on a continuum, e.g. by choosing the indices for the closest point t_i for each $t \in [t_1, t_q]$.



(a) Rel. error plot: CUR and trunc. SVD (b) Gap between theory and practice of the CUR bound in (3.1).

FIG. 3. *Testing the gap between theory and practice for the CUR bound in* (3.1) *using the same set of indices for every parameter value. The parameter-dependent matrix in this example is the same as the one in Section 4.1 (Equation* (3.2)*). The target rank is* $r = 33$ *in this experiment.*

The results are shown in Figure 3. We begin with an initial set of indices and keep using the same set of indices for all other parameter values. The target rank is 33 in this experiment. As shown in Figure 3a, despite using the same set of indices, we achieve a relative error of about 10^{-7} throughout, losing only about 1–2 digits of accuracy compared to the initial accuracy. On the other hand, in Figure 3b, we observe that the bound provided by (3.1) significantly overestimates the true bound. This demonstrates that the quantity $\left\|Q_C(I \cup I_0, :)^\dagger\right\|_2 \left\|V_r(J, :)^{-1}\right\|_2$ alone is insufficient to explain the effectiveness of the CUR decomposition. Such gaps between theoretical bounds and practical performance are common and have been observed in other works, such as [52].

In many problems, δ may be large, yet the CUR decomposition can provide a far better approximation than what is suggested by the bound in (3.1). Moreover, in most practical cases $\|A - [\![A]\!]_r\|_F$ is unknown. This observation motivates us to check whether the previously obtained indices can still be used before recalculating them entirely. To guard against potential large errors, we incorporate error and rank estimation in our algorithms, which we discuss in detail in Sections 3.2 and 3.3.

The second observation is in the role that I_0 plays in (3.1). The set of indices I_0 oversamples the row indices to improve the term $\left\|Q_C(I \cup I_0, :)^\dagger\right\|_2$, benefiting from the fact that $\left\|Q_C(I \cup I_0, :)^\dagger\right\|_2 \leq \left\|Q_C(I, :)^{-1}\right\|_2$, which follows from the Courant-Fischer min-max theorem. Furthermore, when $|I_*| > |J|$, the core matrix $A(I_*, J)$ has larger singular values than when $A(I, J)$ is a square matrix. This improves the accuracy and stability in the computation of the CUR decomposition; see [50] for a detailed discussion. The concept of oversampling has been explored in prior works such as [1, 17, 50, 51, 62] and it is known that oversampling improves the accuracy of the CUR decomposition. In light of this observation, we adopt oversampling, and for definiteness choose to oversample rows in this work, that is, $|I_0| > 0$. We summarize a version of the oversampling algorithm from [50] in Algorithm 3.1, which increases the minimum singular value(s) of $Q_C(I, :)$ by finding unchosen indices that enrich the trailing singular subspace of $Q_C(I, :)$.

Algorithm 3.1 obtains extra row indices I_0, distinct from I, by trying to increase the minimum singular value of $Q_C(I, :)$. The algorithm does this by first projecting

Algorithm 3.1 Oversampling for CUR

Input: $A \in \mathbb{R}^{m\times n}$, column indices J, row indices I, with $|I| = |J| = r < m, n$, oversampling parameter $p(\leq r)$
Output: Extra indices I_0 with $|I_0| = p$

$I_0 = \texttt{OS}(A, I, J, p)$
1: $[Q_C, \sim] = \texttt{qr}\,(A(:, J))$,
2: $[\sim, \sim, V] = \texttt{svd}(Q_C(I, :))$,
3: Set $V_{-p} = V(:, r - p + 1 : r)$, the trailing p right singular vectors of $Q_C(I, :)$.
4: Set $Q_{-C} = Q_C([m] - I, :)V_{-p}$,
5: Apply CPQR on Q_{-C}^T. Let I_0 be the p extra row pivots.

the trailing singular subspace of $Q_C(I, :)$ onto Q_C and choosing, through pivoting, the unchosen indices that contribute the most to the desired subspace, thereby increasing the minimum singular value of $Q_C(I, :)$. The details along with its motivation using cosine-sine decomposition can be found in [50]. The complexity of Algorithm 3.1 is $\mathcal{O}(mr^2 + nrp)$ where the dominant costs come from computing the QR decomposition $\mathcal{O}(mr^2)$ in line 1 and the cost of matrix-matrix multiply $\mathcal{O}(nrp)$ in line 4.

3.1. Computing indices from scratch. In the next two sections, we discuss our algorithms, AdaCUR and FastAdaCUR. For both algorithms, we need to start with an index set for the CUR decomposition. This can sometimes be done offline, for example, if we are solving a Matrix PDE, we can compute the initial set of indices for the CUR decomposition from the initial condition matrix. In this work, we use Algorithm 2.2 to get a set of indices from scratch. The procedure is to first approximate the $(\epsilon/\sqrt{n})$-rank of the initial matrix $A(t_1)$ using randomized rank estimation and then apply pivoting on a random sketch with the estimated $(\epsilon/\sqrt{n})$-rank to get the initial set of indices. Here, we approximate the $(\epsilon/\sqrt{n})$-rank to ensure a relative error of ϵ for the CUR decomposition in the Frobenius norm. More specifically, the $1/\sqrt{n}$ factor arises from the worst case scenario where the trailing singular values of $A(t)$ are all equal. In such cases, the $(\epsilon/\sqrt{n})$-rank is required to guarantee a relative accuracy of ϵ in the Frobenius norm. If additional information about the spectrum of $A(t)$ is available–for instance, if $A(t)$ exhibits rapid singular value decay–then $\epsilon/\sqrt{n}$ can be replaced by $C\epsilon$, where $C < 1$ is a constant. However, to guarantee accuracy, we use $(\epsilon/\sqrt{n})$-rank throughout work. The procedure for computing indices from scratch will be used for getting an initial set of indices for AdaCUR and FastAdaCUR, but also in AdaCUR when the error becomes too large that we need to recompute the indices altogether from scratch.

3.2. AdaCUR algorithm for accuracy. AdaCUR is an algorithm aimed at efficiently computing an accurate low-rank approximation of a parameter-dependent matrix with a given error tolerance. An overview of AdaCUR goes as follows. The algorithm starts by computing the initial set of indices for $A(t_1)$ as discussed in Section 3.1. For subsequent parameter values, i.e., $A(t_j)$ for $j = 2, ..., q$, we first verify whether the indices obtained from the previous parameter value are able to meet some given error tolerance, using an estimate of $\|A - CU^\dagger R\|_F$. If so, we continue to the next parameter value. Should the indices fail to meet the tolerance, we make low-cost minor modifications to the indices and test whether the adjustments satisfy the error tolerance. If this is still unsuccessful, we recompute the indices entirely from scratch and move onto the next parameter value. Minor modifications include adding a small fixed number of indices, reordering them based on their importance and removing some if necessary. The details are in the following paragraph. The addition

and removal of indices, as well as recomputation of indices using rank estimation if necessary, make the algorithm rank-adaptive and computing the relative error at each instance make the algorithm certifiable. AdaCUR is presented in Algorithm 3.2.

Algorithm 3.2 AdaCUR with certified accuracy

Input: Parameter-dependent matrix $A(t) \in \mathbb{R}^{m\times n}$ $(m \geq n)$, evaluation points $t_1, ..., t_q$, error tolerance ϵ, error sample size s (rec. $s = 5$), oversampling parameter p.

Output: Matrices $C(t_j), R(t_j), U(t_j)$ defining a subset of columns and rows of $A(t_j)$ and their intersection for $j = 1, ..., q$.

1: $[I, J] = \texttt{Rand_Pivot_RankEst}(A(t_1), \epsilon/\sqrt{n})$ ▷ Algorithm 2.2
2: $I_0 = \texttt{OS}(A(t_1), I, J, p)$ ▷ Algorithm 3.1
3: Set $C(t_1) = A(t_1)(:, J)$, $R(t_1) = A(t_1)(I \cup I_0, :)$ and $U(t_1) = A(t_1)(I \cup I_0, J)$,
4: **for** $j = 2, ..., q$ **do**
5: Draw a Gaussian embedding $\Gamma_s \in \mathbb{R}^{s\times m}$
6: Sketch $X_s = \Gamma_s A(t_j) \in \mathbb{R}^{s\times n}$
7: Set $E_s = X_s - \Gamma_s A(t_j)(:, J)A(t_j)(I \cup I_0, J)^\dagger A(t_j)(I \cup I_0, :)$ ▷ Sketch of the residual
8: Approximate relative error $E = \|E_s\|_F / \|X_s\|_F$
9: **if** $E > \epsilon$ **then**
10: $[I_1, J_1] = \texttt{Pivot_Residual}(E_s, I, J)$ ▷ Algorithm 2.4
11: Append indices $I \leftarrow I \cup I_1$ and $J \leftarrow J \cup J_1$
12: $[\sim, \sim, I_2] = \texttt{sRRQR}\left(A(t_j)\,(I \cup I_0, J)^T\right)$ [a]
13: $[\sim, R, J_2] = \texttt{sRRQR}\,(A(t_j)\,(I \cup I_0, J))$
14: $r = \left|\left\{R_{ii} : |R_{ii}| > \epsilon/\sqrt{n} \cdot |R_{11}|\right\}\right|$ ▷ Estimate relative $(\epsilon/\sqrt{n})$-rank
15: $I \leftarrow (I \cup I_0)\,(I_2(1 : r))$ ▷ Select r important row indices from $I \cup I_0$
16: $I_0 \leftarrow (I \cup I_0)\,(I_2(r + 1 : r + p))$ ▷ Select the next p important row indices from $I \cup I_0$
17: $J \leftarrow J\,(J_2(1 : r))$ ▷ Select r important column indices from J
18: Set $E_s = X_s - \Gamma_s A(t_j)(:, J)A(t_j)(I \cup I_0, J)^\dagger A(t_j)(I \cup I_0, :)$
19: Recompute relative error $E = \|E_s\|_F / \|X_s\|_F$
20: **if** $E > \epsilon$ **then**
21: $[I, J] = \texttt{Rand_Pivot_RankEst}(A(t_j), \epsilon/\sqrt{n})$ ▷ Recompute indices (Algorithm 2.2)
22: $I_0 = \texttt{OS}(A(t_j), I, J, p)$ ▷ Algorithm 3.1
23: Set $C(t_j) = A(t_j)(:, J)$, $R(t_j) = A(t_j)(I \cup I_0, :)$ and $U(t_j) = A(t_j)(I \cup I_0, J)$.

[a] $[Q, R, I] = \texttt{sRRQR}(A)$ performs strong rank-revealing QR factorization [27] on A where Q is an orthonormal matrix, R is an upper triangular matrix and I is a set of pivots.

AdaCUR (Algorithm 3.2) takes the parameter-dependent matrix $A(t)$, evaluation points $t_1, ..., t_q \in D$, error tolerance ϵ, error sample size s, and oversampling parameter p as input. The algorithm produces the low-rank factors $C(t_j), R(t_j), U(t_j)$ as output such that the factors are a subset of columns, rows and their intersection of $A(t_j)$, respectively, and $A(t_j) \approx C(t_j)U(t_j)^\dagger R(t_j)$. If the entries of the parameter-dependent matrix can be computed quickly afterwards, the algorithm can be modified to output only row and column indices, saving storage. AdaCUR is quite involved, so we break it down into two core parts:

- Section 3.2.1: Error estimation using previous indices (lines 5–8),
- Section 3.2.2: Low-cost minor modifications (lines 10–19).

The other lines involve computing the indices from scratch, which follow the same discussion as Section 3.1.

3.2.1. Initial error estimation. In the for loop, the first task is to quantify how well the previous set of indices perform on the current parameter value. Error estimation is used for this task where we generate a Gaussian embedding Γ_s in line 5, sketch the matrix corresponding to the current parameter value $X_s = \Gamma_s A(t_j)$ in line 6 and the residual error matrix for the CUR decomposition in line 7, and finally, estimating the relative error E using the sketches in line 8. The sketch of the residual

error matrix E_s is used for two tasks: approximating the relative error (lines 8, 19) and computing additional indices to reduce the relative error (line 10). If the relative error E is less than the tolerance ϵ, we use the previous set of indices and store the corresponding columns, rows and their intersection in line 23, after which we continue to the next parameter value t_j. On the other hand, if the relative error exceeds the tolerance, we make low-cost minor modifications in an attempt to reduce the relative error, as we describe next.

3.2.2. Low-cost minor modifications. The low-cost modifications involve trying to enlarge the index set using the sketch of the residual error matrix E_s (lines 10–11), reordering them by importance (lines 12–13), removing unimportant indices (lines 14–17), and recomputing the relative error (lines 18–19). More specifically, in line 10, randomized pivoting is used on the sketch of the residual matrix, E_s as in Algorithm 2.4 to obtain an additional set of s row and column indices. We append the extra set of s indices to the original sets of indices I and J in line 11, and order the indices in terms of their importance in lines 12–13. Here, Gu-Eisenstat's strong rank-revealing QR factorization [27] is used, but we can use other strong rank-revealing algorithms or more expensive algorithms with strong theoretical guarantees such as [14, 49], because we are working with a small $O(r) \times r$ matrix $A(t_j)(I \cup I_0, J)$. In line 14, we approximate the $(\epsilon/\sqrt{n})$-rank of $A(t_j)(I \cup I_0, J)$ by looking at the diagonal entries of the upper triangular factor in sRRQR as they give an excellent approximation to the singular values of the original matrix.[8] In lines 15–17, we truncate based on the rank computed in line 14. The ordering of the indices is important here. We recompute the relative error in lines 18–19,[9] and if the relative error E is smaller than the tolerance, we store the new set of columns, rows of $A(t_j)$ and their intersection in line 23 and continue to the next parameter value. If the relative error exceeds the tolerance, there are two recourses. AdaCUR outlines the first option in lines 21–22 where we recompute the indices altogether from scratch. In line 21, we perform both randomized rank estimation and randomized pivoting for two reasons. First, randomized rank estimation is extremely cheap when we have the row sketch already and second, when there is a sudden large change in rank, we can use randomized rank estimation to adjust the rank efficiently; note that when minor modifications did not help in lines 10–17, the matrix may have changed drastically.

An alternative to recomputing the indices. An alternative approach, not included in AdaCUR, is to increase the error sample size s gradually by incrementing its value, say $2s, 4s$, and so on, and updating the sketch accordingly by appending to it. As s increases, the number of additional important indices obtained from randomized pivoting in line 10 also increases, which will eventually reduce the relative error to less than the tolerance ϵ.

Role of the error sample size s. The value of s can be important for two reasons: error estimation and minor modifications. If we set s to be larger, we obtain higher accuracy in our error estimation at the cost of more computation. However, we obtain a larger set of extra indices for the minor modification. This can help the algorithm

[8] Instead of approximating the $(\epsilon/\sqrt{n})$-rank of $A(t_j)(I \cup I_0, J)$, we can set the rank tolerance slightly smaller, for example, $0.5\epsilon/\sqrt{n}$, to account for the approximation error and decrease the chance of recomputing the indices from scratch (lines 21–22) by keeping slightly more indices than necessary.

[9] For theoretical guarantee, the Gaussian embedding Γ_s needs to be independent from the modified indices [26], however the extra set of indices is dependent on Γ_s as we apply pivoting on the sketched residual in line 10. Nonetheless, the dependency is rather weak so Γ_s can be reused without losing too much accuracy.

avoid the if statement in line 20, which recomputes the indices from scratch; see also Section 4.3. We can also create a version with increasing s if minor modifications fail to decrease the relative error below the tolerance, as described in the final part of the previous paragraph.

Complexity. Let r_j be the size of the index set for parameter t_j, i.e., the rank of the CUR decomposition for $A(t_j)$, Q_1 the set of parameter values for which we recompute the indices from scratch, i.e., invoked lines $1, 2, 21, 22$, and Q_2 the set of parameter values for which we do not need to recompute the indices from scratch. Note that $t_1 \in Q_1$, $Q_1 \cup Q_2 = \{t_1, ..., t_q\}$ and $Q_1 \cap Q_2 = \emptyset$. Then the complexity of AdaCUR is

(3.3)
$$\mathcal{O}\Big(\sum_{j \in Q_1} \big(r_j T_{A(t_j)} + (m+n) r_j^2 + n r_j (r_j + p) \big) + \sum_{j \in Q_2} \big(s T_{A(t_j)} + (m+n) r_j s + nsp \big) \Big)$$

where $T_{A(t_j)}$ is the cost of matrix-vector multiply with $A(t_j)$ or $A(t_j)^T$. The dominant cost, which can be quadratic with respect to m and n, comes from sketching, which cost $\mathcal{O}(r_j T_{A(t_j)})$ in lines $1, 21$ and $\mathcal{O}(s T_{A(t_j)})$ in line 6. The dominant linear costs come from pivoting and oversampling in lines $1, 2, 21, 22$, which cost $\mathcal{O}((m+n) r_j^2 + n r_j p)$ and computing the sketch of the residual error matrix in lines $7, 18$, which is $\mathcal{O}(m r_j s + ns(r_j + p))$. All the other costs are smaller linear costs or sublinear with respect to m and n, for example, the SRTT cost in randomized rank estimation in lines $1, 21$ is $\mathcal{O}(n r_j \log r_j)$ and the cost of sRRQR in lines $12, 13$ is $\mathcal{O}((r_j + p + s)(r_j + s)^2)$. When $s, p = \mathcal{O}(1)$ the complexity simplifies to

$$\mathcal{O}\left(\sum_{j \in Q_1} \big(r_j T_{A(t_j)} + (m+n) r_j^2 \big) + \sum_{j \in Q_2} \big(T_{A(t_j)} + (m+n) r_j \big) \right). \tag{3.4}$$

It turns out that in many applications, the cardinality of the set Q_1 is small when we choose the oversampling parameter p and the error sample size s appropriately, making the algorithm closer to $\mathcal{O}(T_{A(t_j)})$ for the jth parameter value; see Sections 4.2 and 4.3. This makes AdaCUR usually faster than other existing methods such as dynamical low-rank approximation [41], and randomized SVD and generalized Nyström method [37]; see Section 4.5.2.

3.3. FastAdaCUR algorithm for speed. The bottleneck in AdaCUR is usually in the sketching, which is crucial for reliably computing the set of row and column indices for the CUR decomposition and for error estimation, ensuring the algorithm's robustness. Without sketching, our algorithm would have linear complexity with respect to m and n; see (3.4). In this section, we propose FastAdaCUR that achieves linear complexity after the initial phase of computing the initial set of indices for $A(t_1)$. FastAdaCUR offers the advantage that the number of rows and columns required to be seen for each parameter value is approximately its target rank. Therefore, there is no need to read the whole matrix, which is advantageous in settings where the entries are expensive to compute. Unfortunately, the fast version does suffer from adversarial examples as FastAdaCUR does not look at the entire matrix; see Section 4.4. Nevertheless, the algorithm seems to perform well in practice for easier problems as illustrated in Section 4.

An overview of FastAdaCUR goes as follows. The algorithm starts by computing the initial set of indices for $A(t_1)$ as discussed in Section 3.1. Then for each subsequent parameter value, we first form the core matrix $U_j = A(t_j)(I, J)$ where I and J include

extra indices from the buffer space and oversampling from the previous step $j-1$. Here, the buffer space is the extra indices kept to quickly detect a possible change in rank or the important indices as we iterate through the parameter values. We then order the indices by importance and compute the $(\epsilon/\sqrt{n})$-rank of the core matrix U_j.[10] The order of indices is important here as we explain in (ii) below. If the rank increased from the previous iteration, we add extra indices from the buffer space and replenish it by invoking the oversampling algorithm using $\texttt{OS}(A, I, J, \delta_r)$ (Algorithm 3.1) where δ_r is the rank increase. Conversely, if the rank decreases, we simply remove some indices. Finally, we store the corresponding rows and columns of $A(t_j)$ and their intersection and move to the next parameter value. FastAdaCUR is presented in Algorithm 3.3.

Algorithm 3.3 FastAdaCUR for speed

Input: Parameter-dependent matrix $A(t) \in \mathbb{R}^{m\times n}$ $(m \geq n)$, evaluation points $t_1, ..., t_q$, buffer space b, oversampling parameter p, rank tolerance ϵ.

Output: Matrices $C(t_j), R(t_j), U(t_j)$ defining a subset of columns and rows of $A(t_j)$ and their intersection for $j = 1, ..., q$.

1: $[I, J] = \texttt{Rand_Pivot_RankEst}(A(t_1), \epsilon/\sqrt{n})$ ▷ Algorithm 2.2
2: Set $r \leftarrow |I|$
3: $I_0 = \texttt{OS}(A(t_1), I, J, p + b)$ ▷ Algorithm 3.1; includes buffer space and oversampling
4: $J_0 = \texttt{OS}(A(t_1)^T, J, I, b)$ ▷ Algorithm 3.1; includes buffer space
5: Set $I \leftarrow I \cup I_0$ and $J \leftarrow J \cup J_0$
6: Set $C(t_1) = A(t_1)(:, J(1:r)), R(t_1) = A(t_1)(I(1:r+p), :), U(t_1) = A(t_1)(I(1:r+p), J(1:r))$
7: **for** $j = 2, ..., q$ **do**
8: Set $U = A(t_j)(I, J) \in \mathbb{R}^{(r+b+p)\times(r+b)}$
9: $[\sim, \sim, I_1] = \texttt{sRRQR}\left(U^T\right)$
10: $[\sim, R, J_1] = \texttt{sRRQR}\left(U\right)$
11: $r_0 = \left|\{R_{ii} : |R_{ii}| > \epsilon/\sqrt{n} \cdot |R_{11}|\}\right|$ ▷ Estimate relative $(\epsilon/\sqrt{n})$-rank
12: **if** $r_0 \leq r$ **then**
13: Set $I \leftarrow I\left(I_1(1:r_0+b+p)\right)$ ▷ Reorder row indices by importance and truncate
14: Set $J \leftarrow J\left(J_1(1:r_0+b)\right)$ ▷ Reorder column indices by importance and truncate
15: **else**
16: $I_2 = \texttt{OS}(A(t_j), I, J, r_0 - r)$ ▷ Replenish row indices
17: $J_2 = \texttt{OS}(A(t_j)^T, J, I, r_0 - r)$ ▷ Replenish column indices
18: Set $I \leftarrow I(I_1) \cup I_2$ and $J \leftarrow J(J_1) \cup J_2$ ▷ Reorder and add indices
19: Set $r \leftarrow r_0$ ▷ Update $(\epsilon/\sqrt{n})$-rank
20: Set $C(t_j) = A(t_j)(:, J(1:r)), R(t_j) = A(t_j)(I(1:r+p), :), U(t_j) = A(t_j)(I(1:r+p), J(1:r))$

The main distinction between AdaCUR and FastAdaCUR lies in the existence of error estimation. In AdaCUR, an error estimate is computed for each parameter value, which ensures that the row and the column indices are guaranteed to be good with high probability. This is absent in FastAdaCUR. Consequently, if there is a sudden change in rank or the previous set of indices are unimportant for the current parameter value, FastAdaCUR may deliver poor results. To mitigate this disadvantage, FastAdaCUR incorporates a buffer space (i.e., additional indices in I, J) as input to assist with detecting changes in rank. FastAdaCUR involves many heuristics, so we break it down to discuss and justify each part of the algorithm. We divide FastAdaCUR into three parts:

- Section 3.3.1: Rank estimation using the core matrix (lines 8–11),
- Section 3.3.2: Rank decrease via truncation (lines 12–14),
- Section 3.3.3: Rank increase via oversampling (lines 15–18).

[10] We compute the $(\epsilon/\sqrt{n})$-rank of the core matrix to aim for a relative low-rank approximation error of ϵ in the Frobenius norm. As in AdaCUR, we can set the rank tolerance slightly smaller, say $0.5\epsilon/\sqrt{n}$ to account for the approximation error.

The initial set of indices are constructed from scratch as in Section 3.1, with the indices for the buffer space and oversampling obtained using the oversampling algorithm from Algorithm 3.1.

3.3.1. Rank estimation using the core matrix. Upon entering the for loop in line 7, we first compute the core matrix $U = A(t_j)(I, J) \in \mathbb{R}^{(r+b+p)\times(r+b)}$ (line 8) for the current parameter value, which includes extra indices from the buffer space and oversampling. As FastAdaCUR prioritizes efficiency over accuracy, the core matrix is a sensible surrogate to approximate the singular values of $A(t_j)$ without looking at the entire matrix. The extra indices from the buffer space and oversampling will assist with the approximation as the buffer space allows the core matrix to approximate more singular values, helping to detect a potential $(\epsilon/\sqrt{n})$-rank change. The additional set of row indices from oversampling improves the accuracy of the estimated singular values of $A(t_j)$ (line 10) by the Courant-Fischer min-max theorem, which is used for approximating the relative $(\epsilon/\sqrt{n})$-rank in line 11. In addition, oversampling increases the singular values of the core matrix, thereby allowing FastAdaCUR to increase the rank when appropriate and reduce the error. However, we see in the experiments (Section 4.2) that the role that oversampling plays in FastAdaCUR is rather complicated. We use sRRQR to order the columns and rows of the core matrix by importance in lines 9 and 10, and subsequently estimate the $(\epsilon/\sqrt{n})$-rank of the core matrix U using the diagonal entries of the upper triangular factor in sRRQR in line 11. We use the $(\epsilon/\sqrt{n})$-rank to aim for a tolerance of ϵ in the low-rank approximation. If the computed rank, r_0 is smaller than the previous rank, r, then we decrease the index set via truncation. Otherwise, we increase the index set via oversampling. The details are discussed below.

3.3.2. Rank decrease via truncation. After estimating the $(\epsilon/\sqrt{n})$-rank r_0, if r_0 has not increased from the previous iteration, we adjust the number of indices (either decrease or stay the same) in lines 13 and 14 by applying the order of importance computed using sRRQR in lines 9 and 10, and truncating, if necessary. Specifically, we truncate the trailing $r - r_0$ row and column indices. At the end of this procedure, the algorithm ensures $|I| = r_0 + b + p$ and $|J| = r_0 + b$.

3.3.3. Rank increase via oversampling. If the rank has increased, i.e. $r_0 > r$, we refill the extra indices by the amount the rank has increased by, i.e., $r_0 - r$. Unlike in AdaCUR, for which we have the sketched residual matrix, we do not have a good proxy to obtain a good set of extra indices. Therefore, we use the already-selected columns and rows to obtain an extra set of column and row indices.[11] We use the oversampling algorithm (Algorithm 3.1) to achieve this as it only requires the selected rows and columns as input. Adding extra indices using oversampling has been suggested before in [17] to make small rank increases in the context of dynamical low-rank approximation. We get the extra indices from oversampling in lines 16 and 17 and append to the original set of indices in line 18. At the end of this procedure, the algorithm ensures $|I| = r_0 + b + p$ and $|J| = r_0 + b$.

FastAdaCUR relies on heuristics to make sensible decisions for estimating the $(\epsilon/\sqrt{n})$-rank, selecting the important indices, and adding indices as we describe above. Despite FastAdaCUR being susceptible to adversarial examples, as we see later in Section 4.4, it should work on easier problems. For example, if the singular vectors of

[11] The only recourse to obtaining a reliable set of extra indices is to view the entire matrix; see [43, § 17.5] for a related discussion. However, this puts us in a similar setting as AdaCUR, which we recommend when ensuring accuracy is the priority rather than speed.

$A(t)$ are incoherent [44, § 9.6] or (approximately) Haar-distributed, FastAdaCUR will perform well because uniformly sampled columns and rows approximate $A(t)$ well [11]. In such cases, rank adaption is also expected to work well, as the submatrix of $A(t)$ with oversampling behaves similarly to the two-sided sketch used for randomized rank estimation [45]; see Section 2.3. Additionally, with the aid of buffer space, FastAdaCUR is expected to detect rank changes occurring in the problem and efficiently adapt to the rank changes using oversampling or truncation. Therefore, for problems where $A(t)$ is incoherent for the parameter values of our interest, FastAdaCUR is expected to perform effectively.

Role of the buffer space b. The main role of the buffer space b is to identify rank changes while iterating through the parameter values. A larger value of b enhances the algorithm's ability to accurately detect rank changes at an expense of increased complexity. Furthermore, a larger buffer size allows the algorithm to make larger rank changes as the maximum possible rank change at any point in the algorithm is b; see Section 4.3.

Complexity. Let r_j be the size of the index set for parameter t_j, i.e., the rank of the CUR decomposition for $A(t_j)$. The dominant cost $\mathcal{O}\left(r_1 T_{A(t_1)}\right)$ in FastAdaCUR comes from line 1 for computing the initial set of indices for $A(t_1)$ excluding the buffer space and oversampling. Besides the computation of initial indices, the complexity is at most linear in m and n. For $j = 2, ..., q$, the complexity of FastAdaCUR for the jth parameter value is $\mathcal{O}((m+n)(r_j+p+b)^2)$ if the rank has increased from t_{j-1} to t_j. The cost involves the usage of oversampling to replenish extra indices. If the rank has either decreased or stayed the same then the complexity is $\mathcal{O}((r_j+p+b)(r_j+b)^2)$ for using sRRQR. Hence, excluding the first line, FastAdaCUR runs at most linearly in m and n for each parameter value, making it remarkably fast as demonstrated in Section 4.5.2.

4. Numerical Illustration. In this section, we illustrate AdaCUR (Algorithm 3.2) and FastAdaCUR (Algorithm 3.3) using numerical experiments. The code for the strong rank-revealing QR algorithm [27] is taken from [60]. All experiments were performed in MATLAB version 2021a using double precision arithmetic on a workstation with Intel® Xeon® Silver 4314 CPU @ 2.40GHz (16 cores) and 512GB memory.

In the following sections, we perform different numerical simulations with varying input parameters to test AdaCUR and FastAdaCUR. These simulations allow us to test the following.

1. The accuracy of the approximation obtained from the two algorithms by varying the tolerance ϵ,
2. The role that the oversampling parameter p plays in our algorithms,
3. The role that the error sample size s plays in AdaCUR and the buffer size b in FastAdaCUR,
4. How our algorithms perform on adversarial problems,
5. The speed of our algorithms against existing methods in [37, 41].

It is worth pointing out that in the first three examples $\min_j \|A(t_j) - A(t_{j-1})\|_2 > 0.03$ for all j, indicating that the performance of AdaCUR and FastAdaCUR is not explained by the fact that the matrices are undergoing small perturbations as we iterate through the parameters.

In the experiments, we measure how many times AdaCUR needs to enter some of the 'if' statements, as the heavier computations are usually done inside them. This happens when the current set of indices do not meet the given tolerance in AdaCUR.

We define the following

- h_1 is the number of times AdaCUR has invoked lines 10–18, but not lines 20–21. This happens when only minor modifications were needed to meet the tolerance. The complexity is $\mathcal{O}(T_{A(t_j)} + (m+n)r_j^2)$ for the jth parameter value.
- h_2 is the number of times AdaCUR has invoked lines 20–21. This happens when minor modifications were not enough to meet the tolerance and the row and the column indices had to be recomputed from scratch. The complexity is $\mathcal{O}(r_j T_{A(t_j)} + (m+n)r_j^2)$ for the jth parameter value.

In the plots that depict the $(\epsilon/\sqrt{n})$-rank of the low-rank approximation, a large dot is used to indicate the parameter value for which the indices were recomputed from scratch in AdaCUR. The number of large dots is equal to h_2. In FastAdaCUR, besides the computation of initial sets of indices, the heaviest computation is done when the rank has increased; i.e. in lines 16 and 17 when oversampling is invoked. However, this is heavily dependent on the given problem, so we do not track this. For reference, in FastAdaCUR, a large dot is used to indicate the parameter value where oversampling was applied when the rank increased.

4.1. Varying the tolerance ϵ. In this section, we test the accuracy of our algorithms by varying the tolerance parameter ϵ. We use the same synthetic example [9] used for Figure 3, which is given by

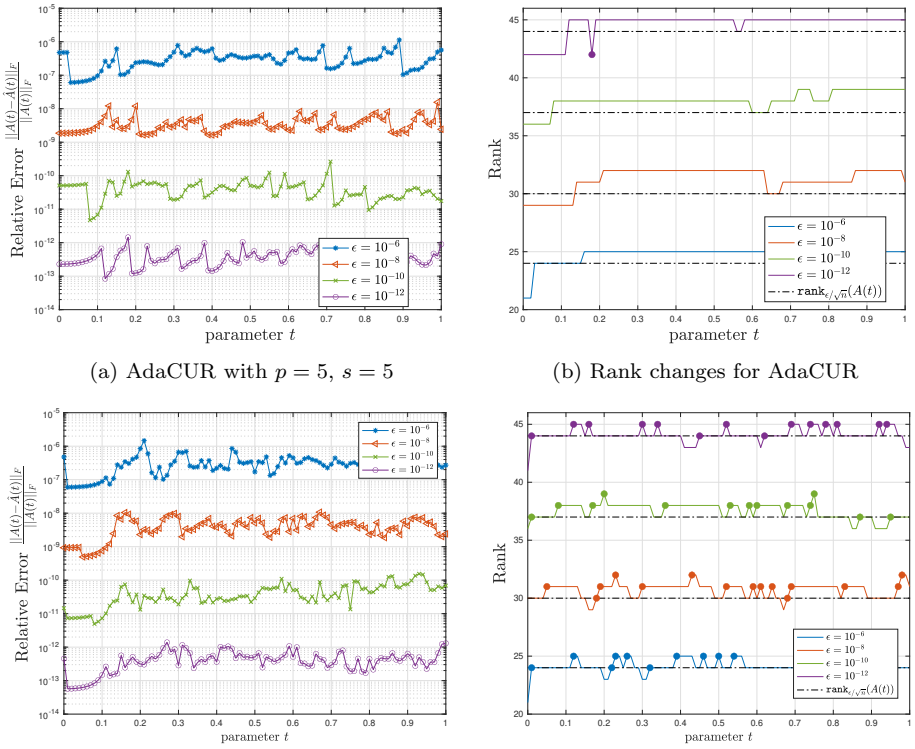
$$A(t) = e^{tW_1} e^t D e^{tW_2}, t \in [0,1]. \tag{3.2}$$

This example tests robustness against small singular values.

The results are depicted in Figure 4 and Table 1. In Figures 4a and 4b, we run AdaCUR with varying tolerance ϵ on the synthetic problem (3.2). We set the oversampling parameter p to 5 and the error sample size s to 5 in this experiment. We first observe in Figure 4a that the relative error for AdaCUR meets the tolerance ϵ within a small constant factor. This is expected, given that the randomized methods in the algorithm typically yield errors within a small constant factor of the optimal solution; therefore one can simply take a smaller ϵ to ensure the actual error is bounded by the desired accuracy. Furthermore, the performance of AdaCUR is reflected in Figure 4b where the algorithm attempts to efficiently find the low-rank approximation of the parameter-dependent matrix by matching the $(\epsilon/\sqrt{n})$-rank at each parameter value. Regarding the algorithm's complexity, as shown in Table 1, we observe that h_1 and h_2 increases as the tolerance ϵ decreases. This increase is anticipated since a higher accuracy requirement generally equates to heavier computations.

In Figures 4c and 4d, we run FastAdaCUR with varying tolerance ϵ on the synthetic problem (3.2). We set the oversampling parameter p to 5 and the buffer space b to 5. We notice in Figure 4c that FastAdaCUR performs well by meeting the tolerance ϵ within a small constant factor. This performance is also reflected in Figure 4d, where the algorithm tries to match the $(\epsilon/\sqrt{n})$-rank at each parameter value. However, as shown in the numerical examples that follow, FastAdaCUR must be used with caution because as the problem becomes more difficult, the error may continue to grow as we iterate through the parameter values.

4.2. Varying the oversampling parameter p. In this part of the section, we test the role that the oversampling parameter p plays in our algorithm. As shown in [50], oversampling can help with the accuracy and the numerical stability of the CUR decomposition. We use a problem based on a PDE [7, 39] whose aim is to solve a heat



(a) AdaCUR with $p = 5$, $s = 5$

(b) Rank changes for AdaCUR

(c) FastAdaCUR with $p = 5$, $b = 5$

(d) Rank changes for FastAdaCUR

FIG. 4. *Testing tolerance parameter ϵ for AdaCUR and FastAdaCUR using the synthetic problem (3.2).*

TABLE 1

The number of instances of heavier computations performed out of 100 *parameter values t by AdaCUR in the experiment corresponding to Figure 4.*

Tolerance ϵ	10^{-6}	10^{-8}	10^{-10}	10^{-12}
Minor mod. only h_1	6	8	10	11
Recomput. Indices h_2	0	0	0	1

equation simultaneously with several heat coefficients. This problem is also known as the parametric cookie problem. In matrix form, the problem is given by

$$\dot{A}(t) = -B_0 A(t) - B_1 A(t) C + b\mathbf{1}^T, \ A(t_0) = A_0 \in \mathbb{R}^{1580\times 101} \tag{4.1}$$

where $B_0, B_1 \in \mathbb{R}^{1580\times 1580}$ are the mass and stiffness matrices, respectively, $b \in \mathbb{R}^{1580}$ is the discretized inhomogeneity and $C = \text{diag}(0, 1, ..., 100)$ is a diagonal matrix containing the parameter samples on its diagonal. We follow [37] by setting $t_0 = -0.01$ and A_0 as the zero matrix and compute the solution at $t = 0$ exactly, after which we discretize the interval $[0, 0.9]$ with 101 uniform points and obtain $A(t)$ using the `ode45` command in MATLAB with tolerance parameters {'RelTol',$1e-12$,'AbsTol',$1e-12$} on each subinterval.

The results are presented in Figure 5 and Table 2. In Figures 5a and 5b, AdaCUR was used with varying oversampling parameter p on the parametric cookie problem

(4.1). We set the tolerance ϵ to 10^{-12} and the error sample size s to 1 in this experiment. In Figure 5a, the algorithm meets the tolerance level up to a modest constant factor while approximately matching the $(\epsilon/\sqrt{n})$-rank in Figure 5b. The benefits of oversampling is demonstrated in Table 2 where the number of instances the algorithm recomputes the indices (which forms the dominant cost), h_2, decreases as the oversampling parameter p increases. This demonstrates that oversampling assists AdaCUR in improving its complexity by reducing the frequency of heavier computations.

In Figures 5c and 5d, we use FastAdaCUR. We vary the oversampling parameter p on the parametric cookie problem (4.1) with the tolerance ϵ equal to 10^{-12} and the buffer size b set to 5. In Figure 5c, the algorithm fails to meet the tolerance $\epsilon = 10^{-12}$ for certain parameter values, but still meets the tolerance within 2 orders of magnitude. By oversampling we are increasing the singular values of the core matrix $U_i = A(t_i)(I, J) \in \mathbb{R}^{(r_i+b+p)\times(r_i+b)}$; see Section 3.3. This, in turn, should advocate a rank increase in FastAdaCUR, which the algorithm uses to reduce the error. However, as the algorithm is based on heuristics to prioritize speed, the impact that the parameters have on the algorithm is rather complicated and inconclusive.[12] Nevertheless, for the accuracy and stability of the CUR decomposition, we recommend at least a little bit of oversampling; see [50].

TABLE 2
The number of instances of heavier computations performed out of 100 *parameter values* t *by AdaCUR in the experiment corresponding to Figure 5.*

Oversampling p	0	5	10
Minor mod. only h_1	24	18	5
Recomput. Indices h_2	6	5	3

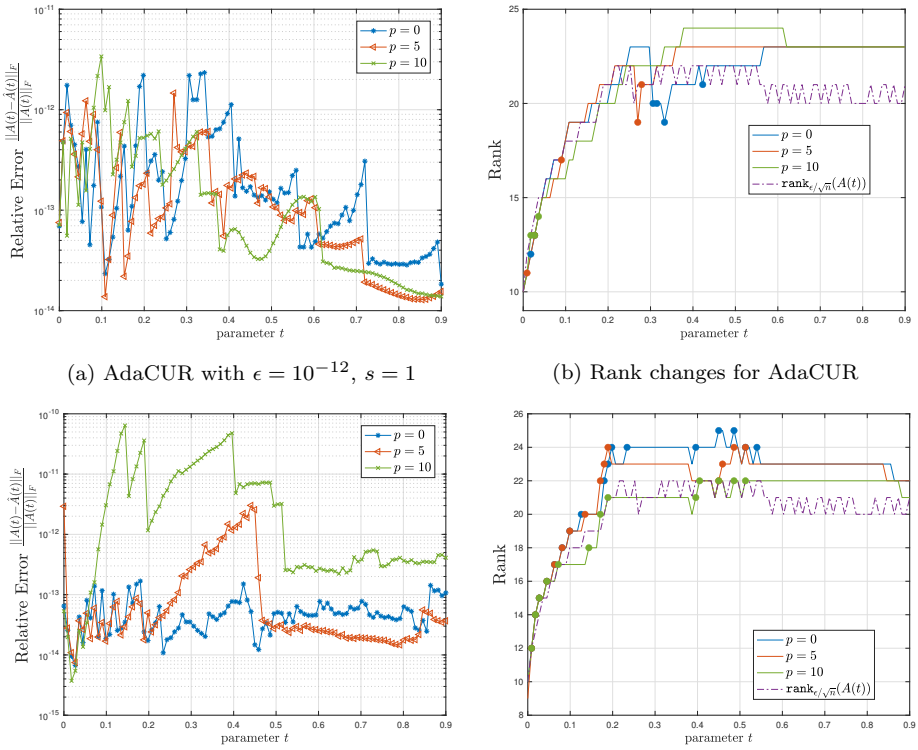
4.3. Varying the error sample size s and the buffer size b. Here we test the role that the error sample size s plays in AdaCUR and the buffer size b in FastAdaCUR. We use the discrete Schrödinger equation in imaginary time for this experiment. The example is taken from [9] and is given by

$$\dot{A}(t) = \frac{1}{2}(D\,A(t) + A(t)\,D) - V_{\cos}\,A(t)\,V_{\cos},\ \ A(0) = A_0, \tag{4.2}$$

where $D = \text{tridiag}(-1, 2, -1)$ is the discrete 1D Laplacian and $V_{\cos}$ is the diagonal matrix with entries $1 - \cos(2j\pi/n)$ for $j = -n/2, ..., n/2 - 1$. We take $n = 512$ and the initial condition A_0 is a randomly generated matrix with singular values 10^{-i} for $i = 1, ..., 512$. We obtain $A(t)$ by discretizing the interval $[0, 0.1]$ with 101 uniform points and using the `ode45` command in MATLAB with tolerance parameters {'RelTol',$1e-12$,'AbsTol',$1e-12$} on each subinterval.

The results are shown in Figure 6 and Table 3. In Figures 6a and 6b, we execute AdaCUR with varying error sample size s on the Schrödinger equation (4.2). The oversampling parameter was set to $p = 10$ and the tolerance $\epsilon = 10^{-12}$. The algorithm satisfies the tolerance ϵ up to a modest constant factor as demonstrated in Figure 6a, while approximately aligning with the $(\epsilon/\sqrt{n})$-rank, as depicted in Figure 6b. As described in the role of the error sample size s in Section 3.2, it makes the minor modifications in AdaCUR more effective. This is demonstrated in Table 3 where the

[12] Running FastAdaCUR on the parametric cookie problem with the same parameter values as in Figure 5c several times yield different results with varying performance between different oversampling values, making the impact that oversampling has for the accuracy inconclusive.



(a) AdaCUR with $\epsilon = 10^{-12}$, $s = 1$

(b) Rank changes for AdaCUR

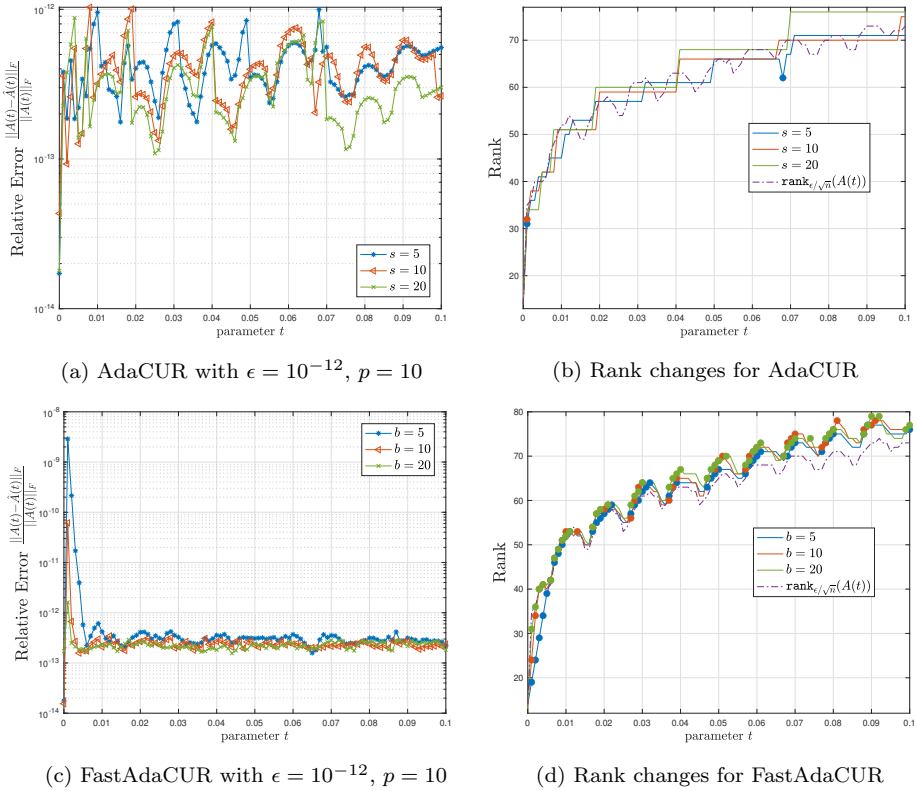
(c) FastAdaCUR with $\epsilon = 10^{-12}$, $b = 5$

(d) Rank changes for FastAdaCUR

FIG. 5. *Testing the oversampling parameter p for AdaCUR and FastAdaCUR using the parametric cookie problem* (4.1).

number of times the algorithm recomputed the indices, h_2, decreases as the error sample size s increases. The experiment demonstrates that with an increase in s, the algorithm becomes better at lowering the relative error using the low-cost minor modifications only. Therefore, the error sample size s should be set sensibly to allow the algorithm to resolve the inaccuracy using minor modifications only so that the algorithm does not recompute indices until it becomes necessary. Note that higher s increases the complexity of AdaCUR, so s should not be too large.

In Figures 6c and 6d, we execute FastAdaCUR. We vary the buffer size b on the Schrödinger equation 4.2 with the tolerance set to $\epsilon = 10^{-12}$ and the oversampling parameter set to $p = 10$. In Figure 6c, the algorithm performs well by meeting the tolerance except for the large spike happening in the first few parameter values. The cause of this spike is explained in Figure 6d where the problem experiences a substantial rank increase in the initial parameter values. Since the algorithm is only able to make a maximum rank increase of b at each iteration, the algorithm struggles to keep up with the large rank increase in the initial parameter values. This observation is evident in Figure 6c, where the initial spike in the graph diminishes as b increases. Therefore, if we expect a large rank increase in the parameter-dependent matrix, the buffer size b should be set to a higher value for those parameter values.



(a) AdaCUR with $\epsilon = 10^{-12}$, $p = 10$

(b) Rank changes for AdaCUR

(c) FastAdaCUR with $\epsilon = 10^{-12}$, $p = 10$

(d) Rank changes for FastAdaCUR

FIG. 6. *Testing the error sample size s for AdaCUR and the buffer size b for FastAdaCUR using the Schrödinger equation* (4.2).

TABLE 3

The number of instances of heavier computations performed out of 100 *parameter values t by AdaCUR in the experiment corresponding to Figure 6.*

Error sample size s	5	10	20
Minor mod. only h_1	12	8	6
Recomput. Indices h_2	2	1	0

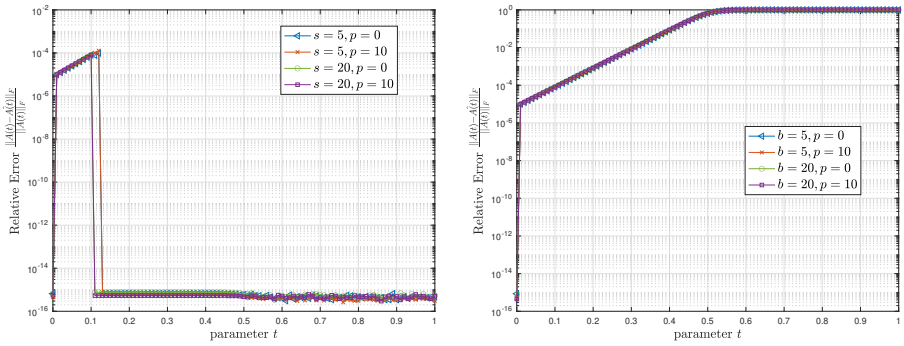
4.4. Adversarial example for FastAdaCUR. In this section, we create an adversarial example for FastAdaCUR to show that the algorithm can fail. We propose the following block diagonal matrix

$$A(t) = \begin{bmatrix} A_1 & 0 & 0 \\ 0 & 0 & c(t)tA_2 \end{bmatrix} \in \mathbb{R}^{300\times 100}, t \in [0,1] \tag{4.3}$$

where $A_1 = \texttt{randn}(100, 20)$, $A_2 = \texttt{randn}(200, 10)$ and $c(t) = 10^{-5+10t}$ using the `randn` command in MATLAB. The parameter-dependent matrix $A(t)$ starts with a large component in the A_1 block, since $c(t)tA_2$ is the zero matrix at $t = 0$. As t increases, the dominant block becomes $c(t)tA_2$.

The results are presented in Figure 7. We execute FastAdaCUR in Figure 7b with tolerance $\epsilon = 10^{-4}$ and various values of buffer size b and oversampling parameter p. FastAdaCUR fails as the error continues to grow as we iterate through the parameter

values. This can be explained by the nature of the algorithm. Since it only considers a submatrix of the original matrix for each parameter value, the $c(t)tA_2$ block remains hidden as t increases. Consequently, the algorithm is never able to capture the $c(t)tA_2$ block, making the approximation poor.[13] On the other hand, as illustrated in Figure 7a, AdaCUR remains successful for the adversarial problem. AdaCUR is able to satisfy the tolerance of 10^{-4}, and when the $c(t)tA_2$ block starts to become dominant, minor modifications or recomputation of indices is employed to capture the $c(t)tA_2$ block, making the algorithm successful.



(a) AdaCUR with $\epsilon = 10^{-4}$ (b) FastAdaCUR with $\epsilon = 10^{-4}$

FIG. 7. *Adversarial example for FastAdaCUR with tolerance* $\epsilon = 10^{-4}$. *The adversarial example causes FastAdaCUR to fail, whereas AdaCUR remains successful.*

4.5. Comparison against other methods. In this section, we test the data-driven aspect of AdaCUR and FastAdaCUR and compare their speed against other methods. Specifically, we benchmark AdaCUR and FastAdaCUR against the randomized SVD and the generalized Nyström method from [37], as well as an algorithm in the dynamical low-rank approximation literature proposed by Lubich and Oseledets [41].

4.5.1. Rank-adaptivity test. To test the rank-adaptive nature of AdaCUR and FastAdaCUR, we use the following synthetic problem. Let $A \in \mathbb{R}^{1000\times 300}$ be a low-rank matrix with $\mathrm{rank}(A) = 50$, Haar-distributed left and right singular vectors, and singular values that decay geometrically from 1 to 10^{-12}. We define the parameter-dependent matrix $A(t)$ as

$$A(i) = A + 10^{-4}\sum_{j=2}^{i} G_1^{(j)} G_2^{(j)} \tag{4.4}$$

where $i \in \{1, 2, ..., 10\}$, and $G_1^{(j)} \in \mathbb{R}^{1000\times 4}$, $G_2^{(j)} \in \mathbb{R}^{4\times 300}$ are Gaussian matrices with i.i.d. entries $\mathcal{N}(0, 1/1000)$ and $\mathcal{N}(0, 1/300)$ respectively. This experiment starts with a rank-50 matrix and incrementally adds rank-4 perturbations of size 10^{-4}.

For AdaCUR and FastAdaCUR, the input parameters were set as follows: $\epsilon = 10^{-9}$ for tolerance, $s, b = 5$ for the error sample size and the buffer size, and $p = 10$ for

[13] This type of counterexample is present in essentially all algorithms that do not view the entire matrix. For example, the algorithm in [17] may suffer from the same adversarial problem as it does not view the entire matrix at each parameter value. See [43, § 17.5] for a related discussion.

the oversampling parameter. For the randomized SVD, generalized Nyström method and the Lubich-Oseledets algorithm, the target rank was set to 50, corresponding to the rank of A. The second sketch size of the generalized Nyström was set to 75. See their respective papers [37, 41] for further details.

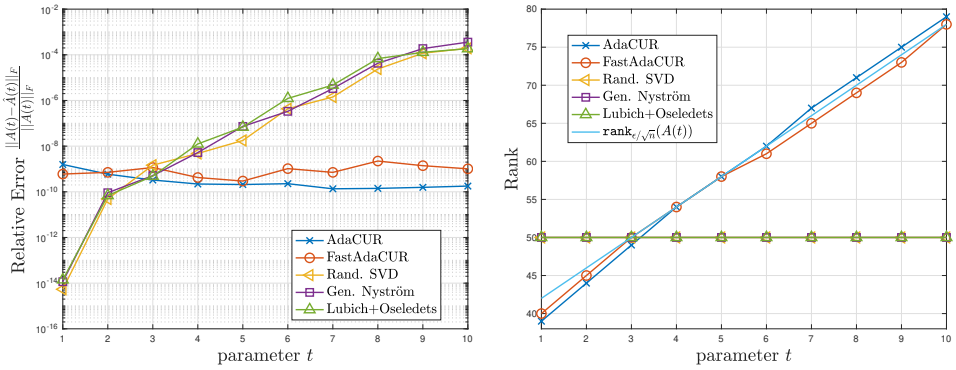


FIG. 8. *Rank-adaptivity test of AdaCUR and FastAdaCUR against the three existing algorithms: randomized SVD, generalized Nyström method [37] and the algorithm by Lubich and Osedelets [41].*

The results are shown in Figure 8. We observe that fixed-rank methods–randomized SVD, generalized Nyström, and the Lubich-Oseledets algorithm–perform poorly as parameter value changes, failing to adapt to evolving data. In contrast, AdaCUR and FastAdaCUR effectively adjust the rank as needed to try maintain the error below the specified tolerance. This is guaranteed for AdaCUR with high probability due to its error control mechanism. This experiment highlights the data-driven nature of AdaCUR and FastAdaCUR, which dynamically adapt to the data, unlike the fixed-rank approaches. While AdaCUR and FastAdaCUR are effective rank-adaptive algorithms, they are not the only methods for addressing rank-adaptivity. Notably, rank-adaptivity has been explored in the dynamical low-rank approximation literature [8, 31, 32, 33]. This experiment highlights the simplicity and effectiveness of AdaCUR and FastAdaCUR in adapting to rank changes.

4.5.2. Speed test. When the target rank r becomes sufficiently large, we anticipate that our algorithms, which run with $\mathcal{O}(T_{A(t)})$ and $\mathcal{O}((m+n)r^2)$ complexity, will outperform many existing methods, which run with $\mathcal{O}(rT_{A(t)})$ complexity. We demonstrate this through experiments using the following artificial problem. Let A be a low-rank matrix with $\text{rank}(A) = 500$ given by $A = U\Sigma V^T \in \mathbb{R}^{50000\times 5000}$ where $U \in \mathbb{R}^{50000\times 500}$ and $V \in \mathbb{R}^{5000\times 500}$ are Haar-distributed orthonormal matrices and $\Sigma \in \mathbb{R}^{500\times 500}$ is a diagonal matrix with entries that decay geometrically from 1 to 10^{-8}. The parameter-dependent matrix $A(t)$ is then defined by

$$A(i) = A + \delta \sum_{j=1}^{i-1} X_j \tag{4.5}$$

where $i \in \{1, 2, ..., 101\}$, $\delta = 10^{-12}$ and X_j's are sparse random matrices generated using $\texttt{sprandn}(50000, 5000, 10^{-5})$ command in MATLAB. The parameter-dependent matrix $A(t)$ starts as a low-rank matrix at $i = 1$, with sparse noise introduced at discrete time intervals. AdaCUR and FastAdaCUR's input parameters were $\epsilon =$

10^{-6} for tolerance, $s, b = 10$ for the error sample size and the buffer size, and the oversampling parameter was set to $p = 10$. The target rank for the randomized SVD, generalized Nyström method and the algorithm by Lubich and Osedelets was set to the rank of A, which is 500. The second sketch size of the generalized Nyström was set to 750. See their respective papers for details [37, 41].

The results are illustrated in Figure 9. For AdaCUR and FastAdaCUR, we notice a slight rise at the beginning, stemming from the initial computation of indices. However, the low-rank approximation of the subsequent parameter values is computed very quickly, making the slope of the cumulative runtime graph relatively flat as we iterate through the parameter values. The graph is notably flat for FastAdaCUR, as it runs at most linearly in m and n at each iteration. On the other hand, the three existing methods display a steeper slope due to the higher computational cost associated with each parameter value. This higher cost stems from the increased number of matrix-vector products with $A(t)$ or its transpose. AdaCUR is approximately 3–5 times faster than the three existing algorithms, while FastAdaCUR is approximately 7–13 times faster than the three existing algorithms.

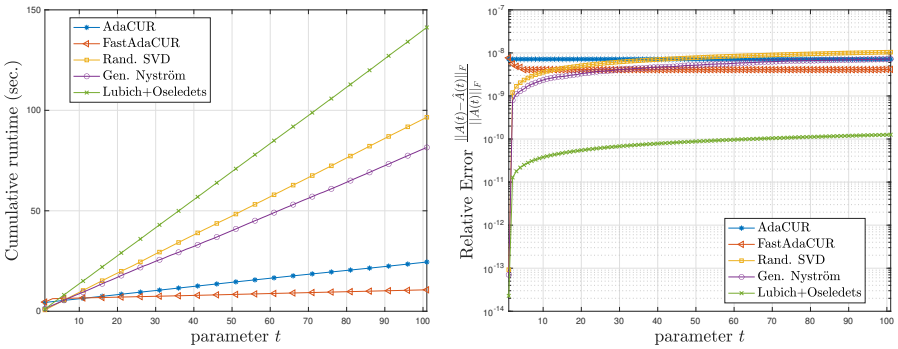


FIG. 9. *Runtime test for AdaCUR and FastAdaCUR against the three existing algorithms: randomized SVD, generalized Nyström method [37] and the algorithm by Lubich and Osedelets [41].*

5. Conclusion. In this work, we devised two efficient rank-adaptive algorithms for computing the low-rank approximation of parameter-dependent matrices: AdaCUR and FastAdaCUR. The key idea behind these algorithms is to try to reuse the row and the column indices from the previous iterate as much as possible. AdaCUR comes with many favourable properties such as rank-adaptivity, error-control and a typical complexity of $\mathcal{O}(T_{A(t)})$, while FastAdaCUR, which is also rank-adaptive, is faster with a complexity that is at most linear in m and n, but lacks error control, making it susceptible to adversarial problems. Nonetheless, FastAdaCUR should work on easier problems such as parameter-dependent matrices that are coherent at each parameter value.

The challenge in FastAdaCUR is that we require an efficient way to detect large errors without viewing the entire matrix. Adversarial examples such as the one in Section 4.4 always exist if we do not view the entire matrix. However, a probabilistic method that allows some sort of error control with high probability would be beneficial for FastAdaCUR. This is left for future work.

REFERENCES

[1] D. Anderson, S. Du, M. Mahoney, C. Melgaard, K. Wu, and M. Gu, *Spectral Gap Error Bounds for Improving CUR Matrix Decomposition and the Nyström Method*, in Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics, G. Lebanon and S. V. N. Vishwanathan, eds., vol. 38 of Proceedings of Machine Learning Research, San Diego, California, USA, 09–12 May 2015, PMLR, pp. 19–27.

[2] A. Andoni and H. L. Nguyên, *Eigenvalues of a matrix in the streaming model*, in Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, 2013, pp. 1729–1737, https://doi.org/10.1137/1.9781611973105.124.

[3] J. Batson, D. A. Spielman, and N. Srivastava, *Twice-ramanujan sparsifiers*, SIAM J. Comput., 41 (2012), pp. 1704–1721, https://doi.org/10.1137/090772873.

[4] C. Boutsidis, P. Drineas, and M. Magdon-Ismail, *Near-optimal column-based matrix reconstruction*, SIAM J. Comput., 43 (2014), pp. 687–717, https://doi.org/10.1137/12086755X.

[5] C. Boutsidis and A. Gittens, *Improved matrix algorithms via the subsampled randomized Hadamard transform*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 1301–1340, https://doi.org/10.1137/120874540.

[6] C. Boutsidis and D. P. Woodruff, *Optimal CUR matrix decompositions*, SIAM J. Comput., 46 (2017), pp. 543–589, https://doi.org/10.1137/140977898.

[7] B. Carrel, M. J. Gander, and B. Vandereycken, *Low-rank parareal: a low-rank parallel-in-time integrator*, BIT, 63 (2023), p. 13, https://doi.org/10.1007/s10543-023-00953-3.

[8] G. Ceruti, J. Kusch, and C. Lubich, *A rank-adaptive robust integrator for dynamical low-rank approximation*, BIT, 62 (2022), pp. 1149–1174, https://doi.org/10.1007/s10543-021-00907-7.

[9] G. Ceruti and C. Lubich, *An unconventional robust integrator for dynamical low-rank approximation*, BIT, 62 (2022), pp. 23–44, https://doi.org/10.1007/s10543-021-00873-0.

[10] S. Chaturantabut and D. C. Sorensen, *Nonlinear model reduction via discrete empirical interpolation*, SIAM J. Sci. Comput., 32 (2010), pp. 2737–2764, https://doi.org/10.1137/090766498.

[11] J. Chiu and L. Demanet, *Sublinear randomized algorithms for skeleton decompositions*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 1361–1383, https://doi.org/10.1137/110852310.

[12] K. L. Clarkson and D. P. Woodruff, *Low-rank approximation and regression in input sparsity time*, J. ACM, 63 (2017), pp. 1–45, https://doi.org/10.1145/3019134.

[13] M. B. Cohen, *Nearly tight oblivious subspace embeddings by trace inequalities*, in Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, 2016, pp. 278–287, https://doi.org/10.1137/1.9781611974331.ch21.

[14] A. Cortinovis and D. Kressner, *Low-rank approximation in the Frobenius norm by column and row subset selection*, SIAM J. Matrix Anal. Appl., 41 (2020), pp. 1651–1673, https://doi.org/10.1137/19M1281848.

[15] M. Dereziński and M. Mahoney, *Determinantal point processes in randomized numerical linear algebra*, Notices Amer. Math. Soc., 60 (2021), p. 1, https://doi.org/10.1090/noti2202.

[16] A. Deshpande, L. Rademacher, S. S. Vempala, and G. Wang, *Matrix approximation and projective clustering via volume sampling*, Theory of Computing, 2 (2006), pp. 225–247, https://doi.org/10.4086/toc.2006.v002a012.

[17] M. Donello, G. Palkar, M. H. Naderi, D. C. Del Rey Fernández, and H. Babaee, *Oblique projection for scalable rank-adaptive reduced-order modelling of nonlinear stochastic partial differential equations with time-dependent bases*, Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 479 (2023), p. 20230320, https://doi.org/10.1098/rspa.2023.0320.

[18] Y. Dong, C. Chen, P.-G. Martinsson, and K. Pearce, *Robust blockwise random pivoting: Fast and accurate adaptive interpolative decomposition*, arXiv preprint arXiv:2309.16002, (2024), https://arxiv.org/abs/2309.16002.

[19] Y. Dong and P.-G. Martinsson, *Simpler is better: a comparative study of randomized pivoting algorithms for CUR and interpolative decompositions*, Adv. Comput. Math., 49 (2023), https://doi.org/10.1007/s10444-023-10061-z.

[20] P. Drineas, M. W. Mahoney, and S. Muthukrishnan, *Relative-error CUR matrix decompositions*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 844–881, https://doi.org/10.1137/07070471X.

[21] Z. Drmač and S. Gugercin, *A new selection operator for the discrete empirical interpolation method—improved a priori error bound and extensions*, SIAM J. Sci. Comput., 38 (2016), pp. A631–A648, https://doi.org/10.1137/15M1019271.

[22] J. A. Duersch and M. Gu, *Randomized projection for rank-revealing matrix factorizations*

and low-rank approximations, SIAM Rev., 62 (2020), pp. 661–682, https://doi.org/10.1137/20M1335571.

[23] P. Y. Gidisu and M. E. Hochstenbach, *A hybrid DEIM and leverage scores based method for CUR index selection*, in Progress in Industrial Mathematics at ECMI 2021, M. Ehrhardt and M. Günther, eds., Cham, 2022, Springer International Publishing, pp. 147–153, https://doi.org/10.1007/978-3-031-11818-0_20.

[24] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, 4 ed., 2013.

[25] S. Goreinov, E. Tyrtyshnikov, and N. Zamarashkin, *A theory of pseudoskeleton approximations*, Linear Algebra Appl., 261 (1997), pp. 1–21, https://doi.org/https://doi.org/10.1016/S0024-3795(96)00301-1.

[26] S. Gratton and D. Titley-Peloquin, *Improved bounds for small-sample estimation*, SIAM J. Matrix Anal. Appl., 39 (2018), pp. 922–931, https://doi.org/10.1137/17M1137541.

[27] M. Gu and S. C. Eisenstat, *Efficient algorithms for computing a strong rank-revealing QR factorization*, SIAM J. Sci. Comput., 17 (1996), pp. 848–869, https://doi.org/10.1137/0917055.

[28] N. Halko, P.-G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev., 53 (2011), p. 217–288, https://doi.org/10.1137/090771806.

[29] K. Hamm and L. Huang, *Perspectives on CUR decompositions*, Appl. Comput. Harmon. Anal., 48 (2020), pp. 1088–1099, https://doi.org/10.1016/j.acha.2019.08.006.

[30] K. Hamm and L. Huang, *Perturbations of CUR decompositions*, SIAM J. Matrix Anal. Appl., 42 (2021), pp. 351–375, https://doi.org/10.1137/19M128394X.

[31] C. D. Hauck and S. Schnake, *A predictor-corrector strategy for adaptivity in dynamical low-rank approximations*, SIAM J. Matrix Anal. Appl., 44 (2023), pp. 971–1005, https://doi.org/10.1137/22M1519493.

[32] J. S. Hesthaven, C. Pagliantini, and N. Ripamonti, *Rank-adaptive structure-preserving model order reduction of hamiltonian systems*, ESAIM: M2AN, 56 (2022), pp. 617–650, https://doi.org/10.1051/m2an/2022013.

[33] M. Hochbruck, M. Neher, and S. Schrammer, *Rank-adaptive dynamical low-rank integrators for first-order and second-order matrix differential equations*, BIT, 63 (2023), p. 9, https://doi.org/10.1007/s10543-023-00942-6.

[34] R. A. Horn and C. R. Johnson, *Matrix Analysis*, Cambridge University Press, 2 ed., 2012, https://doi.org/10.1017/9781139020411.

[35] E. Kieri, C. Lubich, and H. Walach, *Discretized dynamical low-rank approximation in the presence of small singular values*, SIAM J. Numer. Anal., 54 (2016), pp. 1020–1038, https://doi.org/10.1137/15M1026791.

[36] O. Koch and C. Lubich, *Dynamical low-rank approximation*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 434–454, https://doi.org/10.1137/050639703.

[37] D. Kressner and H. Y. Lam, *Randomized low-rank approximation of parameter-dependent matrices*, Numer. Lin. Alg. Appl., (2024), p. e2576, https://doi.org/https://doi.org/10.1002/nla.2576.

[38] D. Kressner, J. Latz, S. Massei, and E. Ullmann, *Certified and fast computations with shallow covariance kernels*, Foundations of Data Science, 2 (2020), pp. 487–512, https://doi.org/10.3934/fods.2020022.

[39] D. Kressner and C. Tobler, *Low-rank tensor Krylov subspace methods for parametrized linear systems*, SIAM J. Matrix Anal. Appl., 32 (2011), pp. 1288–1316, https://doi.org/10.1137/100799010.

[40] C. Lubich, *Low-Rank Dynamics*, Springer International Publishing, Cham, 2014, pp. 381–396, https://doi.org/10.1007/978-3-319-08159-5_19.

[41] C. Lubich and I. V. Oseledets, *A projector-splitting integrator for dynamical low-rank approximation*, BIT, 54 (2014), pp. 171–188, https://doi.org/10.1007/s10543-013-0454-0.

[42] M. W. Mahoney and P. Drineas, *CUR matrix decompositions for improved data analysis*, Proceedings of the National Academy of Sciences, 106 (2009), pp. 697–702, https://doi.org/10.1073/pnas.0803205106.

[43] P.-G. Martinsson, *Fast direct solvers for elliptic PDEs*, SIAM, 2019.

[44] P.-G. Martinsson and J. A. Tropp, *Randomized numerical linear algebra: Foundations and algorithms*, Acta Numer., 29 (2020), p. 403–572, https://doi.org/10.1017/s0962492920000021.

[45] M. Meier and Y. Nakatsukasa, *Fast randomized numerical rank estimation for numerically low-rank matrices*, Linear Algebra Appl., 686 (2024), pp. 1–32, https://doi.org/https://doi.org/10.1016/j.laa.2024.01.001.

[46] Y. Nakatsukasa, *Fast and stable randomized low-rank matrix approximation*, arXiv preprint arXiv:2009.11392, (2020), https://arxiv.org/abs/2009.11392.

[47] J. Nelson and H. L. Nguyên, *OSNAP: Faster numerical linear algebra algorithms via sparser subspace embeddings*, in Proc. IEEE 54th Annu. Symp. Found. Comput. Sci., 2013, pp. 117–126, https://doi.org/10.1109/FOCS.2013.21.

[48] A. Nonnenmacher and C. Lubich, *Dynamical low-rank approximation: applications and numerical experiments*, Mathematics and Computers in Simulation, 79 (2008), pp. 1346–1357, https://doi.org/https://doi.org/10.1016/j.matcom.2008.03.007.

[49] A. Osinsky, *Close to optimal column approximations with a single SVD*, arXiv preprint arXiv:2308.09068, (2023), https://doi.org/10.48550/arXiv:2308.09068.

[50] T. Park and Y. Nakatsukasa, *Accuracy and stability of CUR decompositions with oversampling*, arXiv preprint arXiv:2405.06375, (2024), https://arxiv.org/abs/2405.06375.

[51] B. Peherstorfer, Z. Drmač, and S. Gugercin, *Stability of discrete empirical interpolation and gappy proper orthogonal decomposition with randomized and deterministic sampling points*, SIAM J. Sci. Comput., 42 (2020), pp. A2837–A2864, https://doi.org/10.1137/19M1307391.

[52] D. C. Sorensen and M. Embree, *A DEIM induced CUR factorization*, SIAM J. Sci. Comp., 38 (2016), pp. A1454–A1482, https://doi.org/10.1137/140978430.

[53] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*, SIAM, 1997, https://doi.org/10.1137/1.9780898719574.

[54] J. A. Tropp, *Improved analysis of the subsampled randomized Hadamard transform*, Advances in Adaptive Data Analysis, 03 (2011), p. 115–126, https://doi.org/10.1142/s1793536911000787.

[55] J. A. Tropp, A. Yurtsever, M. Udell, and V. Cevher, *Practical sketching algorithms for low-rank matrix approximation*, SIAM J. Matrix Anal. Appl., 38 (2017), p. 1454–1485, https://doi.org/10.1137/17m1111590.

[56] M. Udell and A. Townsend, *Why are big data matrices approximately low rank?*, SIAM Journal on Mathematics of Data Science, 1 (2019), pp. 144–160, https://doi.org/10.1137/18M1183480.

[57] S. Voronin and P.-G. Martinsson, *Efficient algorithms for CUR and interpolative matrix decompositions*, Adv. Comput. Math., 43 (2017), pp. 495–516, https://doi.org/10.1007/s10444-016-9494-8.

[58] D. Woodruff, *Sketching as a Tool for Numerical Linear Algebra*, Foundations and Trends® in Theoretical Computer Science Series, Now Publishers, 2014.

[59] F. Woolfe, E. Liberty, V. Rokhlin, and M. Tygert, *A fast randomized algorithm for the approximation of matrices*, Appl. Comput. Harmon. Anal., 25 (2008), pp. 335–366, https://doi.org/10.1016/j.acha.2007.12.002.

[60] X. Xing, *Strong rank revealing QR decomposition*, 2024, https://www.mathworks.com/matlabcentral/fileexchange/69139-strong-rank-revealing-qr-decomposition. Retrieved December 24, 2023.

[61] N. L. Zamarashkin and A. I. Osinsky, *On the existence of a nearly optimal skeleton approximation of a matrix in the Frobenius norm*, Dokl. Math., 97 (2018), pp. 164–166, https://doi.org/10.1134/S1064562418020205.

[62] R. Zimmermann and K. Willcox, *An accelerated greedy missing point estimation procedure*, SIAM J. Sci. Comput., 38 (2016), pp. A2827–A2850, https://doi.org/10.1137/15M1042899.