LU factorization with errors[*]

Jean-Guillaume Dumas
Université Grenoble Alpes
Laboratoire Jean Kuntzmann, CNRS, UMR 5224
38058 Grenoble, cedex 9, France
Jean-Guillaume.Dumas@univ-grenoble-alpes.fr

Joris van der Hoeven
CNRS
Laboratoire d'informatique de l'École polytechnique
LIX, UMR 7161 CNRS, 91120 Palaiseau, France
vdhoeven@lix.polytechnique.fr

Clément Pernet
Université Grenoble Alpes
Laboratoire Jean Kuntzmann, CNRS, UMR 5224
38058 Grenoble, cedex 9, France
Clement.Pernet@univ-grenoble-alpes.fr

Daniel S. Roche
United States Naval Academy
Annapolis, Maryland, U.S.A.
roche@usna.edu

ABSTRACT

We present new algorithms to detect and correct errors in the lower-upper factorization of a matrix, or the triangular linear system solution, over an arbitrary field. Our main algorithms do not require any additional information or encoding other than the original inputs and the erroneous output. Their running time is softly linear in the dimension times the number of errors when there are few errors, smoothly growing to the cost of fast matrix multiplication as the number of errors increases. We also present applications to general linear system solving.

KEYWORDS

Error correction; matrix factorization; linear algebra; sparse interpolation; algorithms

ACM Reference Format:
Jean-Guillaume Dumas, Joris van der Hoeven, Clément Pernet, and Daniel S. Roche. 2019. LU factorization with errors. In *Proceedings of ACM ISSAC (ISSAC'19),* Manuel Kauers (Ed.). ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The efficient detection and correction of computational errors is an increasingly important issue in modern computing. Such errors can result from hardware failures, communication noise, buggy software, or even malicious servers or communication channels.

The first goal for fault-tolerant computing is verification. Freivalds presented a linear-time algorithm to verify the correctness of a single matrix product [18]. Recently, efficient verification algorithms for a wide range of computational linear algebra problems have been developed [14–16, 24].

Here we go further and try to correct any errors that arise. This approach is motived by the following scenarios:

Large scale distributed computing. In high-performance computing, the failure of some computing nodes (fail stop) or the corruption of some bits in main memory by cosmic radiation (soft errors) become relevant. The latter type can be handled by introducing redundancy and applying classical error correction, either at the hardware level (e.g., with ECC RAM), or integrated within the computation algorithm, as in many instances of Algorithm Based Fault Tolerance (ABFT) [4, 9, 11, 22].

Outsourcing computation. When running some computation on one or several third-party servers, incorrect results may originate from either failure or malicious corruption of some or all of the third parties. The result obtained is then considered as an approximation of the correct result.

Intermediate expression swell. It often happens that symbolic computations suffer from requiring large temporary values, even when both the input and output of a problem are small or sparse. It can then be more efficient to use special methods that are able to determine or guess the sparsity pattern and exploit this in the computation. Sparse polynomial and rational function interpolation has been developed [2, 23, 31, 33] as such a technique. The connection with error correction comes from the fact that we may regard a sparse output as the perturbation of some trivial approximate solution, such as zero or the identity matrix.

Fault-tolerant computer algebra. Some recent progress on fault tolerant algorithms has been made for Chinese remaindering [3, 21, 28], system solving [5, 25], matrix multiplication and inversion [20, 30, 32], and function recovery [8, 26].

1.1 Our setting

In this paper, we focus on LU-factorization and system solving. We will assume the input (such as a matrix A to be factored) is known, as well as an *approximate output* (such as a candidate LU-factorization) which may contain errors. We seek efficient algorithms to recover the correct outputs from the approximate ones.

Our work can be seen as part of a wider effort to understand how techniques for fault tolerance without extra redundancy extend beyond basic operations such as matrix multiplication. It turns out that the complexities for other linear algebra problems are extremely sensitive to the precise way they are stated. For instance, in the case of system solving, the complexities for error-correction

[*] This work is partly funded by the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541) and the French National Research Agency program (ANR-15-IDEX-02).

ISSAC'19, July 2019,
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

are quite different if we assume the LU-factorization to be returned along with the output, or not. The LU-factorization process itself is very sensitive to pivoting errors; for this reason we will assume our input matrix to admit a generic rank profile (GRP).

From an information theoretic point of view, it should be noted that all necessary input information in order to compute the output is known to the client. The approximate output is merely provided by the server in order to make the problem computationally cheaper for the client. In theory, if the client has sufficient resources, he could perform the computation entirely by himself, while ignoring the approximate output. Contrary to what happens in the traditional theory of error correcting codes, it is therefore not required to transmit the output data in a redundant manner (in fact, we might even not transmit anything at all).

In our setting, unlike classical coding theory, it is therefore more appropriate to measure the correction capacity in terms of the amount of computational work for the decoding algorithm rather than the amount of redundant information. We also put no *a priori* restriction on the number of errors: ideally, the complexity should increase with the number of errors and approach the cost of the computation without any approximate output when the number of errors is maximal.

As a consequence, the error-correcting algorithms we envision can even be used in an extreme case of the second scenario: a malicious third party might introduce errors according to patterns that are impossible to correct using traditional bounded distance decoding approaches. Concerning the third scenario, the complexity of our error-correcting algorithms is typically sensitive to the sparsity of a problem, both in the input *and* output.

Another general approach for error correction is discussed in Section 2: if we allow the server to communicate some of the intermediate results, then many linear algebra operations can be reduced in a simple way to matrix multiplication with error correction.

1.2 General outline and main results

General notations. Throughout our paper, F stands for an effective field with $\#\mathsf{F} \in \mathbb{N} \cup \{\infty\}$ elements. We write $\mathsf{F}^{m \times n}$ for the set of $m \times n$ matrices with entries in F and $\#M$ for the number of non-zero entries of a matrix $M \in \mathsf{F}^{m \times n}$. The soft-Oh notation $\tilde{O}(\ldots)$ is the same as the usual big-Oh notation but ignoring sub-logarithmic factors: $f = \tilde{O}(g)$ if and only if $f \in O(g(\log g)^{O(1)})$ for cost functions f and g.

Let ω be a constant between 2 and 3 such that two $n \times n$ matrices can be multiplied using $O(n^{\omega})$. In practice, we have $\omega = 3$ for small dimensions n and $\omega \leqslant \log_2 7 \approx 2.81$ for larger dimensions, using Strassen multiplication (the precise threshold depends on the field F; Strassen multiplication typically becomes interesting for n of the order of a few hundred). The best asymptotic algorithm currently gives $\omega < 2.3728639$ [19]. From a practical point of view, the ω notation indicates whether an algorithm is able to take advantage of fast low-level matrix multiplication routines.

Generic solution. In Section 2, we first describe a generic strategy for fault-tolerant computations in linear algebra. However, this strategy may require the server to communicate certain results of intermediate computations. In the remainder of the paper, we only consider a stronger form of error correction, for which only the final results need to be communicated.

LU-factorization. Let $A \in \mathsf{F}^{n \times n}$ be an invertible matrix with generic rank profile (GRP). Let $\mathcal{L}, \mathcal{U} \in \mathsf{F}^{n \times n}$ be (resp.) lower- and upper-triangular matrices that are "close to" the LU-factorization of A: $A \approx \mathcal{L}\mathcal{U}$. Specifically, suppose there exist sparse upper- and lower-triangular error matrices $E, F \in \mathsf{F}^{n \times n}$ with

$$A = (\mathcal{L} + E)(\mathcal{U} + F), \qquad \#E + \#F \leqslant k.$$

In the following we write the (possibly) faulty matrices in calligraphic fonts. We specify that $\mathcal{L}$ has 1's on the diagonal, so that E is strictly lower-triangular whereas F may have corrections on the diagonal or above it.

Given $A, \mathcal{L}, \mathcal{U}$, the goal is to determine the true $L = \mathcal{L} + E$ and $U = \mathcal{U} + F$ as efficiently as possible. Our main result is a probabilistic Monte Carlo algorithm for doing this in time

$$\tilde{O}\Big(t + \min\{kn, k^{\omega-2} n^{4-\omega}\}\Big),$$

for any constant probability of failure, where t is the number of nonzero terms in all input matrices, k is the number of errors in $\mathcal{L}$ or $\mathcal{U}$, and n is the matrix dimension (see Theorem 3). Here we measure the complexity in terms of he number of operations in F, while assuming that random elements in F can be generated with unit cost. Note that because the number of errors k cannot exceed the total dense size n^2, this complexity is never larger than $\tilde{O}(n^{\omega})$, which is the cost of re-computing the LU-factorization with no approximate inputs, up to logarithmic factors. In Section 3.4, we also show how to generalize our algorithm to rectangular, possibly rank deficient matrices (still under the GRP assumption).

System solving. In Section 4, we turn to the problem of solving a linear system $XA = B$, where A is as above and B is also given. Given only possibly-erroneous solution $\mathcal{X}$ to such a system, we do not know of any efficient method to correct the potential errors. Still, we will show how errors can efficiently be corrected if we require the server to return a few extra intermediate results.

More precisely, if B has few rows, then we require an approximate LU-factorization $A \approx \mathcal{L}\mathcal{U}$ and an approximate solution to the triangular system $\mathcal{Y}\mathcal{U} = B$. If B has many rows (with respect to the number of columns), then we require an approximate LU-factorization $A \approx \mathcal{L}\mathcal{U}$ and an approximate inverse $\mathcal{R}$ of $\mathcal{U}$. Given these data, we give algorithms to correct all errors in similar running time as above, depending on the number of errors in all provided inputs.

2 GENERIC SOLUTION

In [24, § 5] a generic strategy is described for verifying the result of a linear algebra computation. The complexity of the verification is the same as the complexity of the actual computation under the assumption that matrix products can be computed in time $\tilde{O}(n^2)$. In this section, we show that a similar strategy can be used to correct errors.

Let $\mathcal{A}$ be any algorithm which uses matrix multiplication. We assume that $\mathcal{A}$ is deterministic; any random bits needed in the computation should be pre-generated by the client and included with the input. The server runs algorithm $\mathcal{A}$ and, each time it performs any matrix multiplication, it adds that product to a list. This list of all intermediate products is sent back to the client. We assume that there may be a small number of errors in any of the intermediate products, but that these errors do not propagate; that is, the total number of erroneous entries in any intermediate product is bounded by some k.

Next, to correct errors, the client also runs algorithm $\mathcal{A}$, except that every time it needs to multiply matrices, it performs error correction instead, using the intermediate result sent by the server. All other operations (additions, comparisons, etc.) are performed directly by the client. The total cost for the server is the same as the normal computation of $\mathcal{A}$ without error correction. The cost for the client, as well as the communication, is the cost that the algorithm *would have* if matrix multiplication could be performed in $O(n^2)$ time. In particular, typical block matrix algorithms such as LU factorization admit a worst case complexity of $\tilde{O}(n^\omega)$ for the server and only $\tilde{O}(n^2)$ for the client (including communications).

The goal of this paper is to perform better, for instance when k is a bound on the number of errors only in L and U and not of all intermediate computations.

3 BLOCK RECURSIVE ALGORITHM

We will now describe an error cleaning algorithm emulating the steps of an LU decomposition algorithm, where each computation task is replaced by an error cleaning task.

The cleaning algorithm to be used needs to satisfy the following properties:

1. it has to be a block algorithm, gathering most operations into matrix multiplications where error cleaning can be efficiently performed by means of sparse interpolation, as in [30, 32];
2. it has to be recursive in order to make an efficient usage of fast matrix multiplication.
3. each block operation must be between operands that are submatrices of either the input matrix A or the approximate $\mathcal{L}$ and $\mathcal{U}$ factors. Indeed, the only data available for the error correction are these three matrices: the computation of any intermediate results that cannot directly be extracted from these matrices would be too expensive for achieving the intended complexity bounds.

In the large variety of LU decomposition algorithms, iterative and block iterative algorithms range in three main categories depending on the scheduling of the update operation (see [10, § 5.4] and [12] for a description): right-looking, left-looking and Crout.

The right-looking variant updates the trailing matrix immediately after a pivot or a block pivot is found, hence generating blocks containing intermediate results, not satisfying (3). Differently, the left-looking and the Crout variants proceed by computing directly the output coefficients in L and U following a column shape (left-looking) or arrow-head (Crout) frontier. Figure 1 summarizes these 3 variants by exposing the memory access patterns of one iteration.

The left-looking and the Crout schedules consist in delaying the computation of the Gauss updates until the time where the elimination front deals with the location under consideration. This is precisely satisfying Condition (3). However these two schedules are usually described in an iterative or block iterative setting. To the best of our knowledge, no recursive variant has been proposed so far. We introduce in Section 3.1 a recursive Crout LU decomposition algorithm on which we will base the error-correction algorithm of section 3.3. Interestingly, we could not succeed in writing a recursive version of a left-looking algorithm preserving Condition (3).

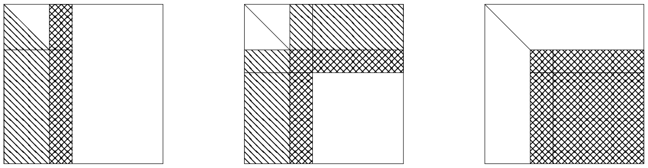

Figure 1: Access pattern of the left-looking (left), Crout (center) and right-looking variants of an LU factorization. Diagonal stripes represent read-only accesses, crossed stripes read-write accesses.

We emphasize that, although our eventual error correction algorithm will follow the recursive Crout variant, no assumption whatsoever is made on the algorithm used (e.g., by a remote server) to produce the approximate LU decomposition used as input.

3.1 Recursive Crout LU decomposition

Algorithm 1 is a presentation of a Crout recursive variant of Gaussian elimination, for a generic rank profile (GRP) matrix. It incorporates the delayed update schedule of the classical block iterative Crout algorithm [10, 12] into a recursive algorithm dividing both row and column dimensions. Note that due to the delayed updates, the recursive algorithm need to be aware of the coefficients in L and U previously computed, hence the entirety of the working matrix has to be passed as input, contrarily to most recursive algorithms [6, 17]. In a normal context, this algorithm could work in-place, overwriting the input matrix A with both factors L and U as the algorithm proceeds.

In our error-correcting context, the input will eventually consist not only of A but also of the approximate $\mathcal{L}$ and $\mathcal{U}$. Therefore, in Algorithm 1 below, we treat the matrix A as an unmodified input and fill in the resulting L and U into a separate matrix M. In Algorithm 3, this matrix M will initially contain the approximate $\mathcal{L}$ and $\mathcal{U}$ which will be overwritten by the correct L and U.

The main work of the Crout decomposition consists of dot products (in the base case), matrix multiplications and triangular solves. We define TRSM as a triangular system solve with matrix right-hand side, with some variants depending whether the triangular matrix is lower ('L') or upper ('U') triangular, and whether the triangular matrix is on the left ('L', for $X \leftarrow T^{-1}A$) or on the right ('R', for for $X \leftarrow AT^{-1}$). These always work in-place, overwriting the right-hand side with the solution. For instance:

- $\texttt{URTrsm}(A, U)$ is a right solve with upper-triangular U which transforms A to A' such that $A'U = A$.
- $\texttt{LLTrsm}(L, B)$ is a left solve with lower-triangular L which transforms B to B' such that $LB' = B$.

In the algorithms, we use the subscript ♣ to denote "indices 2 and 3", so for example $n_\clubsuit = n_2 + n_3$ and

$$A_{\clubsuit\clubsuit} = \left[\begin{array}{c:c} A_{22} & A_{23} \\ \hdashline A_{32} & A_{33} \end{array}\right].$$

Theorem 1. *$Crout(M, A, 0, n)$ overwrites M with a complete LU factorization of $A \in \mathsf{F}^{n\times n}$ in $O(n^\omega)$ operations.*

Algorithm 1 `Crout`$(M, A_{\clubsuit\clubsuit}, n_1, n_\clubsuit)$

```
Require: $M = \left[\begin{array}{c|c} L_{11}\backslash U_{11} & U_{1\clubsuit} \\ \hline L_{\clubsuit 1} & \end{array}\right]$ is $(n_1 + n_\clubsuit) \times (n_1 + n_\clubsuit)$
Require: $L_{11}$ (resp. $U_{11}$) is unit lower (resp. upper) triangular
Require: $A_{\clubsuit\clubsuit}$ is $n_\clubsuit \times n_\clubsuit$
Require: $A = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{1\clubsuit} \\ L_{\clubsuit 1}U_{11} & A_{\clubsuit\clubsuit} \end{bmatrix}$ has GRP
Ensure: $M = \left[L\backslash U\right]$ such that $A = L{\cdot}U$
 1: if $n_\clubsuit = 1$ then
 2:     $U_{\clubsuit\clubsuit} \longleftarrow A_{\clubsuit\clubsuit} - L_{\clubsuit 1}{\cdot}U_{1\clubsuit}$        ▷ dot product
                                                        ▷ $L_{\clubsuit\clubsuit}$ is implicitly [1]
 3: else
 4:     Decompose $n_\clubsuit = n_2 + n_3$ with $n_2 = \lceil n_\clubsuit/2 \rceil$, $n_3 = \lfloor n_\clubsuit/2 \rfloor$
 5:     Split $M = \left[\begin{array}{c|c|c} L_{11}\backslash U_{11} & U_{12} & U_{13} \\ \hline L_{21} & & \\ \hline L_{31} & & \end{array}\right]$
 6:     Split $A_{\clubsuit\clubsuit} = \left[\begin{array}{c|c} A_{22} & A_{23} \\ \hline A_{32} & A_{33} \end{array}\right]$
 7:     Crout $\left(\left[\begin{array}{c|c} L_{11}\backslash U_{11} & U_{12} \\ \hline L_{21} & \end{array}\right], A_{22}, n_1, n_2\right)$
 8:     Here $M = \left[\begin{array}{c|c|c} L_{11}\backslash U_{11} & U_{12} & U_{13} \\ \hline L_{21} & L_{22}\backslash U_{22} & \\ \hline L_{31} & & \end{array}\right]$
 9:     $U_{23} \longleftarrow A_{23} - L_{21}U_{13}$
10:     LLTrsm$(U_{23}, L_{22})$                        ▷ $L_{22}U_{23} = A_{23} - L_{21}U_{13}$
11:     $L_{32} \longleftarrow A_{32} - L_{31}U_{12}$
12:     URTrsm$(L_{32}, U_{22})$                        ▷ $L_{32}U_{22} = A_{32} - L_{31}U_{12}$
13:     Here $M = \left[\begin{array}{cc|c} L_{11}\backslash U_{11} & U_{12} & U_{13} \\ L_{21} & L_{22}\backslash U_{22} & U_{23} \\ \hline L_{31} & L_{32} & \end{array}\right]$
14:     Crout $(M, A_{33}, n_1 + n_2, n_3)$
15:     Here $M = \begin{bmatrix} L_{11}\backslash U_{11} & U_{12} & U_{13} \\ L_{21} & L_{22}\backslash U_{22} & U_{23} \\ L_{31} & L_{32} & L_{33}\backslash U_{33} \end{bmatrix}$
16: end if
```

Proof. Correctness is proven by induction on $n_\clubsuit$. For $n_\clubsuit = 1$ we have

$$\begin{bmatrix} L_{11} & 0 \\ L_2 & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{1\clubsuit} \\ 0 & A_{\clubsuit\clubsuit} - L_{\clubsuit 1}U_{1\clubsuit} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{1\clubsuit} \\ L_{\clubsuit 1}U_{11} & A_{\clubsuit\clubsuit} \end{bmatrix}.$$

For $n_\clubsuit > 1$, we have

$$LU = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} & L_{11}U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} & L_{21}U_{13} + L_{22}U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33} \end{bmatrix}.$$

After the first recursive call in step 7, we have $L_{21}U_{12} + L_{22}U_{22} = A_{22}$. In step 10, we ensured that $A_{23} = L_{22}U_{23} + L_{21}U_{13}$. Similarly, $A_{32} = L_{32}U_{22} + L_{31}U_{12}$ from step 12. Finally, after the second recursive call, we have $A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33}$. This concludes the proof that $A = LU$ at the end of the algorithm.

The complexity bound stems from the fact that the dot products in step 2 cost $O(n^2)$ overall, and that the matrix multiplications and system solves require $O(n^\omega)$ [13]. □

3.2 Error-correcting triangular solves

The cost of Algorithm 1 is dominated by matrix-matrix products of off-diagonal blocks of L and U and triangular solving in steps 9–12. The first task in adapting this algorithm for error correction is to perform error correction in this triangular solving step, treating the right-hand side as an unevaluated *black box matrix* in order to avoid the matrix-matrix multiplication.

We explain the process to correct errors in $\mathcal{U}_{23}$ (an equivalent process works in the transpose for $\mathcal{L}_{32}$): recall steps 9 and 10 of Algorithm 1: $U_{23} \longleftarrow A_{23} - L_{21}U_{13}$ and `LLTrsm`(U_{23}, L_{22}). Mathematically, these steps perform the computation: $U_{23} \longleftarrow L_{22}^{-1}(A_{23} - L_{21}U_{13})$. At this point in the error correction, A_{23} is part of the original input matrix while L_{21}, U_{13}, and L_{22} have already been corrected by the recursive calls.

The idea is to be able to correct the next parts of an approximate $\mathcal{U}$, namely $\mathcal{U}_{23}$, without recomputing it. Algorithm 2 below does this following the approach of [32]: for $k \leqslant mn$ total errors within $c \leqslant n$ erroneous columns, less than $c/2$ columns can have more than $s = \lfloor 2k/c \rfloor$ errors. Therefore, we start with a candidate for k, then try to correct s errors per erroneous column. If k is correct, then they will be corrected in fewer than $\log(c) \leqslant \log(n)$ iterations. If fewer than $c/2$ columns are corrected on some step, this indicates that the guess for k was too low, so we double the guess for k and continue. This is shown in Algorithm 2, where as previously mentioned, a normal font is an already correct matrix block and a calligraphic font denotes a matrix block to be corrected. We also recall that $\#X$ stands for the number of nonzero elements in X, and define `ColSupport` of a matrix M to be the indices of columns of M with any nonzero entries.

Due to the crucial use of sparse interpolation, we require a high-order element θ in the underlying field F. If no such θ exists, we simply replace F by an extension field. The cost of computing in such an extension field will only induce a logarithmic overhead.

Theorem 2. *For a failure bound* $0 < \varepsilon < 1$, $A \in \mathsf{F}^{m \times \ell}$, $B \in \mathsf{F}^{\ell \times n}$, $C \in \mathsf{F}^{m \times n}$, $U \in \mathsf{F}^{n \times n}$, $\mathcal{R} \in \mathsf{F}^{m \times n}$, *with total non-zero entries*

$$t = \max\{\#A, \#B, \#C, \#U, \#\mathcal{R}\} \leqslant (m + n)(\ell + n),$$

and k *errors in* $\mathcal{R}$, *Algorithm 2 is correct and runs in time*

$$\tilde{O}\Big((t + k + m)(1 + \log_{\#\mathsf{F}} \tfrac{1}{\varepsilon}) + \max\{n, \ell\} \cdot \min\{k, k^{\omega-2} n^{3-\omega}\}\Big).$$

Proof. Without loss of generality, we may assume that $m < \#\mathsf{F}$ in step 3. Indeed, in the contrary case, arithmetic in $\mathbb{F}_{q^\nu}$ is only $\tilde{O}(\nu) = \tilde{O}(\log m)$ times more expensive than arithmetic in $\mathbb{F}_q$, which is absorbed by the soft-Oh of the claimed complexity bound.

Define R as the correct output matrix such that $RU = H$ and consider the beginning of some iteration through the loop. Line 12 computes with high probability the column support of the remaining errors $R - (\mathcal{R} + E)$, viewing this as a blackbox $HU^{-1} - \mathcal{R} - E$. We project this blackbox on the left with a block of vectors W using a Freivalds check [18].

Hence P is a $n \times c$ submatrix of a permutation matrix selecting the erroneous columns of $\mathcal{R}$. Selecting the same rows and columns

Algorithm 2 URTrsmEC($\mathcal{R}, H, U, m, \ell, n, \varepsilon$)

```
Require: $\mathcal{R}$ is $m \times n$
Require: $H$ is $m \times n$ presented as an unevaluated blackbox $H = C - AB$ where the inner dimension between $A$ and $B$ is $\ell$
Require: $U$ is $n \times n$ invertible upper triangular
Require: Failure bound $0 < \varepsilon < 1$.
Ensure: $\mathcal{R}$ is updated in-place s.t. $\Pr[\mathcal{R}U = H] \geqslant 1 - \varepsilon$
1: $k \leftarrow 1$                                   ▷ how many errors exist
2: $k' \leftarrow 0$                          ▷ how many errors have been corrected
3: if $m \geqslant \#\mathsf{F}$ then $\mathsf{F} \leftarrow \mathbb{F}_{q^v}$   ($q = \#\mathsf{F}, v = \lceil \log_q(m+1) \rceil$) end if
4: Pick $\theta$ of order $\geqslant m$ in $\mathsf{F}$, with precomputed $(\theta^j)_{0 \leqslant j < m}$
5: $\lambda \leftarrow \left\lceil \log_{\#\mathsf{F}} \left(3n \log_2 n/\varepsilon\right) \right\rceil$
6: $c' \leftarrow 2n$
7: $E \leftarrow 0^{m \times n}$
8: repeat
9:     Pick $W \in \mathsf{F}^{\lambda \times m}$ uniformly at random.
10:    $X \leftarrow WC - (WA)B$                              ▷ $X = WH$
11:    URTrsm($X, U$)                                 ▷ $X = WHU^{-1}$
12:    $(j_1, \dots, j_c) \leftarrow$ ColSupport($X - W\mathcal{R} - WE$)
                                       ▷ columns of $(\mathcal{R} + E)$ with errors
13:    Clear any entries of $E$ from columns $j_1, \dots, j_c$
14:    Update $\mathcal{R} \leftarrow \mathcal{R} + E$,    $k' \leftarrow k' + \#E$
15:    if $c > c'/2$ then $k \leftarrow \max(2k, c)$ end if
                                      ▷ too many errors; $k$ must be wrong
16:    $c' \leftarrow c$
17:    $s \leftarrow \min\left(n, \left\lceil 2\frac{k-k'}{c} \right\rceil\right)$
18:    $V \leftarrow (\theta^{ij})_{0 \leqslant i < 2s, 0 \leqslant j < m}$                 ▷ unevaluated
19:    $P \leftarrow \left[e_{j_1} \quad \dots \quad e_{j_c}\right]$         ▷ selects erroneous cols. of $\mathcal{R}$
20:    $G \leftarrow V(CP) - (VA)(BP) - (V\mathcal{R})(UP)$
21:    URTrsm($G, P^\intercal UP$)              ▷ $GP^\intercal UP = V(H - \mathcal{R}U)P$
22:    Find $S \in \mathsf{F}^{m \times c}$ s.t. $VS = G$ by sparse interpolation
23:    $E \leftarrow SP^\intercal$
24: until $c = 0$
```

in U yields a $c \times c$ matrix $P^\intercal UP$ that is still triangular and invertible. With this, one can form a new blackbox

$$S = (R - \mathcal{R})P = (H - \mathcal{R}U) \cdot P \cdot (P^\intercal UP)^{-1},$$

using the fact that $(R - \mathcal{R})PP^\intercal = (R - \mathcal{R})$. The columns of this blackbox S are viewed as c sparse polynomials whose evaluation at powers of θ are used to recover them via sparse interpolation.

Note that only the columns with at most s nonzero entries are correctly recovered by the sparse interpolation, so some columns of S may still be incorrect. However, any incorrect ones are discovered by the Freivalds check in the next round and never incorporated into $\mathcal{R}$. From the definition of s, and the fact that sparse interpolation works correctly for all s-sparse columns of $R - \mathcal{R}$, we know that every iteration results in either c reducing by half, or k doubling. Therefore the total number of iterations is at most $\log_2 c + \log_2 k \leqslant (1 + 2)\log_2 n = 3\log_2 n$.

According to [32, Lemma 4.1], the probability of failure in each Freivalds check in step 12 is at most ε/n. By the union bound, the probability of failure at *any* of the $\leqslant 3\log_2 n$ iterations is therefore at most ε, as required.

Now for the complexity, the calls to compute the ColSupport on Lines 10 to 12 are all performed using sparse matrix-vector operations, taking $O(\lambda(t + k))$. From [32, Lemma 6.1], the multiplication by the Vandermonde $2s \times n$ matrix V in $V(CP)$, VA, and $V\mathcal{R}$ all take $\tilde{O}(t + k + m)$ operations.

The recovery of S by batched multi-sparse interpolation takes $\tilde{O}(sc + m\log m) = \tilde{O}(k + m)$ operations [32, Theorem 5.2].

What remains are the cost of computing the following:

- The product $(VA)(BP)$, which costs $O(s\ell c/\min\{s, \ell, c\}^{3-\omega})$ using fast matrix multiplication.
- The product $(V\mathcal{R})(UP)$, which costs $O(snc/\min\{s, n, c\}^{3-\omega})$.
- The subroutine URTrsm($G, P^\intercal UP$), which costs the same as it would be to multiply G times $P^\intercal UP$, $O(sc^2/\min\{s, c\}^{3-\omega})$.

From the definition of s we have $sc \in O(k)$. Let $N = \max\{n, \ell\}$. Since $s, c \leqslant n$ and $n, \ell \leqslant N$, all three costs are

$$O(Nk/\min\{s, c\}^{3-\omega}). \tag{1}$$

Until the algorithm terminates, we always have $s, c \geqslant 1$, so (1) is at most $O(Nk)$, proving the first part of the min in the complexity.

For the second part, observe that the number of erroneous columns c must satisfy $k/n \leqslant c \leqslant n$, which means that $\min\{s, c\} \geqslant k/n$ by the definition of s. Then the cost in (1) is bounded above by $O\left(\frac{Nk}{(k/n)^{3-\omega}}\right) \leqslant O\left(N \cdot k^{\omega-2} \cdot n^{3-\omega}\right)$. □

Remark 1. Since the input matrix U is not modified by the algorithm, some entries of U may be defined implicitly — in particular, if the matrix is unit diagonal and the 1's are not explicitly stored.

Remark 2. Transposing the algorithm, we may also correct $L\mathcal{R} = G$, where L is lower triangular:

LLTrsmEC($\mathcal{R}, G, L, n, \ell, m, \varepsilon$) = (URTrsmEC($\mathcal{R}^\intercal, G^\intercal, L^\intercal, m, \ell, n, \varepsilon$))$^\intercal$.

Remark 3. There are two more triangular variants to consider. Computing LRTrsmEC (or, with Remark 2, ULTrsmEC) could be done exactly the same as in Algorithm 2, except that the two subroutine calls, lines 11 and 21, would be to LRTrsm.

3.3 Correcting an invertible LU decomposition

We now have all the tools to correct an LU decomposition. We suppose that the matrix A is non-singular and has generic rank profile (thus there exist unique L and U such that $A = LU$). We are given possibly faulty candidate matrices $\mathcal{L}$, $\mathcal{U}$ and want to correct them: for this we run Algorithm 1, but replace lines 9-12 by two calls to Algorithm 2. The point is to be able to have explicit submatrices of A, L, U, $\mathcal{L}$ and $\mathcal{U}$ for the two recursive calls (no blackbox, nothing unevaluated there), and that all the base cases represent only a negligible part of the overall computations (the base case is the part of the algorithm that is recomputed explicitly when correcting). This is presented in Algorithm 3.

Theorem 3. *For $A \in \mathsf{F}^{n \times n}$ which has GRP, $\mathcal{L} \in \mathsf{F}^{n \times n}$ unit lower triangular and $\mathcal{U} \in \mathsf{F}^{n \times n}$ upper triangular, with total non-zero entries $t = \#A + \#\mathcal{L} + \#\mathcal{U}$, failure bound $0 < \varepsilon < 1$, and k errors in $\mathcal{L}$ and $\mathcal{U}$, Algorithm* CroutEC($\left[\mathcal{L}\backslash\mathcal{U}\right]$, $A, 0, n, \varepsilon$) *is correct and runs in time*

$$\tilde{O}\Big((t + k)(1 + \log_{\#\mathsf{F}} \tfrac{1}{\varepsilon}) + \min\{kn, k^{\omega-2}n^{4-\omega}\}\Big).$$

Algorithm 3 `CroutEC`$(M, A_{\clubsuit\clubsuit}, n_1, n_\clubsuit, \varepsilon)$

```
Require: $M = \left[\begin{array}{c|c} L_{11}\backslash U_{11} & U_{1\clubsuit} \\ \hline L_{\clubsuit 1} & \mathcal{L}_{\clubsuit\clubsuit}\backslash\mathcal{U}_{\clubsuit\clubsuit} \end{array}\right]$ is $(n_1 + n_\clubsuit) \times (n_1 + n_\clubsuit)$
Require: $L_{11}$ (resp. $U_{11}$) is unit lower (resp. upper) triangular
Require: $\mathcal{L}_{\clubsuit\clubsuit}, \mathcal{U}_{\clubsuit\clubsuit}$ are $n_\clubsuit \times n_\clubsuit$ unit lower/upper triangular
Require: $A_{\clubsuit\clubsuit}$ is $n_\clubsuit \times n_\clubsuit$
Require: $A = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{1\clubsuit} \\ L_{\clubsuit 1}U_{11} & A_{\clubsuit\clubsuit} \end{bmatrix}$ has GRP
Require: Failure bound $0 < \varepsilon < 1$
Ensure: $M = \left[L\backslash U\right]$ such that $\Pr[A = L{\cdot}U] \geqslant 1 - \varepsilon$
1: if $n_\clubsuit = 1$ then
2:     $U_{\clubsuit\clubsuit} \leftarrow A_{\clubsuit\clubsuit} - L_{\clubsuit 1}{\cdot}U_{1\clubsuit}$          ▷ dot product
                                   ▷ $L_{\clubsuit\clubsuit}$ is implicitly [1], must be correct
3: else
4:     Decompose $n_\clubsuit = n_2 + n_3$ with $n_2 = \lceil n_\clubsuit/2 \rceil$, $n_3 = \lfloor n_\clubsuit/2 \rfloor$
5:     Split $M = \left[\begin{array}{c|c|c} L_{11}\backslash U_{11} & U_{12} & U_{13} \\ \hline L_{21} & \mathcal{L}_{22}\backslash\mathcal{U}_{22} & \mathcal{U}_{23} \\ \hline L_{31} & \mathcal{L}_{32} & \mathcal{L}_{33}\backslash\mathcal{U}_{33} \end{array}\right]$
6:     Split $A_{\clubsuit\clubsuit} = \left[\begin{array}{c|c} A_{22} & A_{23} \\ \hline A_{32} & A_{33} \end{array}\right]$
7:     CroutEC$\left(\left[\begin{array}{c|c} L_{11}\backslash U_{11} & U_{12} \\ \hline L_{21} & \mathcal{L}_{22}\backslash\mathcal{U}_{22} \end{array}\right], A_{22}, n_1, n_2, \varepsilon/4\right)$
8:     Here $M = \left[\begin{array}{c|c|c} L_{11}\backslash U_{11} & U_{12} & U_{13} \\ \hline L_{21} & L_{22}\backslash U_{22} & \mathcal{U}_{23} \\ \hline L_{31} & \mathcal{L}_{32} & \mathcal{L}_{33}\backslash\mathcal{U}_{33} \end{array}\right]$
9:     LLTrsmEC$(\mathcal{U}_{23}, A_{23} - L_{21}U_{13}, L_{22}, n_2, n_1, n_3, \varepsilon/4)$
                                   ▷ $A_{23} - L_{21}U_{13}$ is left unevaluated
10:    URTrsmEC$(\mathcal{L}_{32}, A_{32} - L_{31}U_{12}, U_{22}, n_3, n_1, n_2, \varepsilon/4)$
                                   ▷ $A_{32} - L_{31}U_{12}$ is left unevaluated
11:    Here $M = \left[\begin{array}{cc|c} L_{11}\backslash U_{11} & U_{12} & U_{13} \\ L_{21} & L_{22}\backslash U_{22} & U_{23} \\ \hline L_{31} & L_{32} & \mathcal{L}_{33}\backslash\mathcal{U}_{33} \end{array}\right]$
12:    CroutEC$(M, A_{33}, n_1 + n_2, n_3, \varepsilon/4)$
13:    Now $M = \begin{bmatrix} L_{11}\backslash U_{11} & U_{12} & U_{13} \\ L_{21} & L_{22}\backslash U_{22} & U_{23} \\ L_{31} & L_{32} & L_{33}\backslash U_{33} \end{bmatrix}$.
14: end if
```

Proof. Correctness follows from Theorems 1 and 2: we rewrite Algorithm 1, but replace the intermediate two matrix multiplications and two `Trsms` by two calls, with blackboxes, to Algorithm 2. Passing $\varepsilon/4$ to all subroutines ensures that the total probability of failure in any Frievalds check in any `TrsmEC` is at most ε. Note that the shrinking ε does not affect the soft-oh complexity, since at the bottom level we will have $\varepsilon' = \varepsilon/(4^{\log_2(n)}) = \varepsilon/n^2$, and $\log \frac{1}{\varepsilon'}$ is $O(\log \frac{1}{\varepsilon} + \log n)$.

For the complexity, note that since A has GRP, $\#A \geqslant n$. The stated cost bound depends crucially on the following fact: in each level of recursive calls to `CroutEC`, each call is correcting a single diagonal block $\mathcal{L}_{\clubsuit\clubsuit}\backslash\mathcal{U}_{\clubsuit\clubsuit}$ and uses only the parts of M and A above and left of that block.

This claim is true by inspection of the algorithm: the blocks $\mathcal{L}_{22}\backslash\mathcal{U}_{22}$ and $\mathcal{L}_{33}\backslash\mathcal{U}_{33}$ being corrected in the two recursive calls to `CroutEC` on Lines 7 and 12 are clearly disjoint diagonal blocks. And we see also that the algorithm never uses the top-left part of M, namely $L_{11}\backslash U_{11}$; all arguments to the calls to `TrsmEC` on Lines 9 and 10 are (disjoint) submatrices above and left of $\mathcal{L}_{\clubsuit\clubsuit}\backslash\mathcal{U}_{\clubsuit\clubsuit}$, as shown, e.g., in Figure 2.

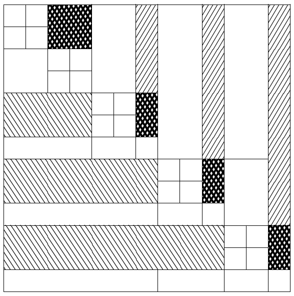

Figure 2: All updates at the third recursion level at step 9. The figure indicates the locations of $\mathcal{U}_{23}$, L_{21}, and U_{13}.

With this understanding, we can perform the analysis. The work of the algorithm is entirely in the dot products in the base case, and the calls to `TrsmEC` in the recursive case.

For the base case dot products, these are to correct the diagonal entries of $\mathcal{U}$, using the diagonal of A and disjoint, already-corrected rows of L and columns of U. Therefore the total cost of the dot products is $O(t + k)$.

For the rest, consider the ith recursive level of calls to `CroutEC`, where $0 \leqslant i < \log_2 n$. There will be exactly 2^{i+1} calls to `TrsmEC` on this level, whose inputs are all disjoint, from the claim above. Write m_{ij}, ℓ_{ij}, etc. for the parameters in the jth call to `TrsmEC` on level i, for $1 \leqslant j \leqslant 2^{i+1}$.

Every call to `TrsmEC` on the ith level satisfies:

- $m_{ij}, n_{ij} \in O(n/2^i)$;
- $\ell_{ij}, N_{ij} = \max\{n_{ij}; \ell_{ij}\} \leqslant n$;
- $\varepsilon_{ij} = \varepsilon/4^i$, so that $\lceil \log_{\#\mathsf{F}}(1/\varepsilon_{ij}) \rceil$ is $O((1 + \log_{\#\mathsf{F}} \frac{1}{\varepsilon}) + \log n)$;
- $\sum_{j=1}^{2^{i+1}} t_{ij} \leqslant t$ because the submatrices are disjoint at the same recursive level; and
- $\sum_{i=0}^{\log_2 n} \sum_{j=1}^{2^{i+1}} k_{ij} \leqslant k$ because each error is only corrected once.

Now we wish to compute the total cost of all calls to `TrsmEC`, which by Theorem 2 and the notation just introduced, is soft-oh of

$$\sum_{i=0}^{\log_2 n} \sum_{j=1}^{2^{i+1}} \left[(t_{ij} + k_{ij} + m_{ij})(1 + \log_{\#\mathsf{F}} \tfrac{1}{\varepsilon_{ij}}) + N_{ij} \cdot \min\{k_{ij}, k_{ij}^{\omega-2} n_{ij}^{3-\omega}\} \right].$$

Taking the first term of the sum, this is

$$\begin{aligned} &\sum_{i=0}^{\log_2 n} \sum_{j=1}^{2^{i+1}} (t_{ij} + k_{ij} + \tfrac{n}{2^i})(1 + \log_{\#\mathsf{F}} \tfrac{1}{\varepsilon} + \log n) \\ &\leqslant O\Big((t \log n + k + 2n \log n)(1 + \log_{\#\mathsf{F}} \tfrac{1}{\varepsilon} + \log n) \Big) \\ &= \tilde{O}\Big((t + k + n)(1 + \log_{\#\mathsf{F}} \tfrac{1}{\varepsilon}) \Big), \end{aligned}$$

which gives the first term in our stated complexity.

The second term of the sum simplifies to

$$n \cdot \sum_{i=0}^{\log_2 n} \sum_{j=1}^{2^{i+1}} \min\{k_{ij}, k_{ij}^{\omega-2}(n/2^i)^{3-\omega}\}. \tag{2}$$

Now observe that, by definition, each individual summand is less than or equal to both parts of the min expression. Therefore we bound the sum of minima by the minimum of sums; that is, the previous summation is at most

$$n \cdot \min\left\{\textstyle\sum_i \sum_j k_{ij}, \quad \sum_i \sum_j k_{ij}^{\omega-2}(n/2^i)^{3-\omega}\right\}.$$

Because each error is only corrected once, $\sum_i \sum_j k_{ij} \leqslant k$. Using Hölder's inequality and $0 \leqslant \omega - 2 < 1$, this yields

$$\sum_{j=1}^{r} k_{ij}^{\omega-2} \leqslant r \left(\frac{\sum_{j=1}^{r} k_{ij}}{r}\right)^{\omega-2} \leqslant r^{3-\omega} k^{\omega-2},$$

for all r. Applying this to the second summation above gives

$$\begin{aligned}\sum_{i=0}^{\log_2 n} \sum_{j=1}^{2^{i+1}} k_{ij}^{\omega-2}(n/2^i)^{3-\omega} &= n^{3-\omega} \sum_{i=0}^{\log_2 n} 2^{(\omega-3)i} \sum_{j=1}^{2^{i+1}} k_{ij}^{\omega-2} \\ &\leqslant n^{3-\omega} \sum_{i=0}^{\log_2 n} 2^{(\omega-3)i} \cdot 2^{(i+1)(3-\omega)} \cdot k^{\omega-2} \\ &\leqslant O(k^{\omega-2} n^{3-\omega} \log n).\end{aligned}$$

Then the entirety of (2) becomes just $\tilde{O}\big(\min\{kn, k^{\omega-2}n^{4-\omega}\}\big)$, which gives the second term in the stated complexity. □

3.4 Correcting a rectangular, rank-deficient LU

For $m \leqslant n$, assume first that $A = [A_1 \quad A_2]$ is a rectangular $m{\times}n$ matrix such that A_1 is square $m{\times}m$ with GRP. Assume also that $[\mathcal{L}\backslash\mathcal{U}_1 \quad \mathcal{U}_2]$ is an approximate LU decomposition. Then we may correct potential errors as follows:

(1) `CroutEC`$([\mathcal{L}_1\backslash\mathcal{U}_1], A_1, 0, n, \varepsilon/2)$; ▹ corrects $L_1\backslash U_1$
(2) `LLTrsmEC`$(\mathcal{U}_2, A_2, L_1, m, m, n-m, \varepsilon/2)$. ▹ corrects U_2

Second, if A_1 is rank deficient, but still GRP, then the first occurrence of a zero pivot $U_{\clubsuit\clubsuit}$, line 2 in Algorithm 3, reveals the correct rank: at this point L_{11}, U_{11}, and the upper (resp. left) part of $L_{\clubsuit 1}$ (resp. $U_{1\clubsuit}$) are correct. It is thus sufficient to stop the elimination there, and recover the remaining parts of

$$L = \begin{bmatrix} L_{11} \\ L_{\clubsuit 1} \end{bmatrix} \in \mathsf{F}^{m\times r}, \quad U = \begin{bmatrix} U_{11} & U_{1\clubsuit} \end{bmatrix} \in \mathsf{F}^{r\times n},$$

using (corrected) triangular system solves, as follows:

(1) Let $r = \operatorname{rank} A$ and $A = \begin{bmatrix} A_{11} & A_{1\clubsuit} \\ A_{\clubsuit 1} & A_{\clubsuit\clubsuit} \end{bmatrix}$, where A_{11} is $r{\times}r$.
(2) `CroutEC`$(\left[\mathcal{L}_{11}\backslash\mathcal{U}_{11}\right], A_{11}, 0, r, \varepsilon/3)$
(3) `LLTrsmEC`$(\mathcal{U}_{1\clubsuit}, A_{1\clubsuit}, L_{11}, r, r, n-r, \varepsilon/3)$. ▹ corrects $U_{1\clubsuit}$
(4) `URTrsmEC`$(\mathcal{L}_{\clubsuit 1}, A_{\clubsuit 1}, U_{11}, m-r, r, r, \varepsilon/3)$. ▹ corrects $L_{\clubsuit 1}$

4 SYSTEM SOLVING

One application of LU factorization is linear system solving. Say A is $n{\times}n$ invertible, and B is $m{\times}n$. Ideally, correcting a solution $\mathcal{X}$ to the linear system $XA = B$, should depend only on the errors in $\mathcal{X}$. Indeed, our algorithm `URTrsmEC` does exactly this in the special case that A is upper-triangular.

Unfortunately, we do not know how to do this for a general non-singular A using the previous techniques, as we for instance do not know of an efficient way to compute the nonzero columns of the erroneous entries in $(X - \mathcal{X}) = BA^{-1} - \mathcal{X}$.

By adding some data to the solution $\mathcal{X}$ (and therefore, unfortunately, potentially more errors), there are several possibilities:

- Generically, a first solution is to proceed as in Section 2. One can use [24], but then the complexity depends not only on errors in $\mathcal{X}$, but on errors in all the intermediate matrix products.
- A second solution is to invert the matrix A using [32, Algorithm 6] (`InverseEC`) and then multiply the right-hand side using [32, Algorithm 5] (`MultiplyEC`). This requires some extra data to correct, namely a candidate inverse matrix, and the complexity depends on errors appearing now both in the system solution and in that inverse candidate matrix as follows:
  (1) $Z \longleftarrow$ `InverseEC`$(A, \mathcal{Z})$; ▹$Z = A^{-1}$
  (2) $X \longleftarrow$ `MultiplyEC`$(B, Z, \mathcal{X})$. ▹$X = BA^{-1}$

Now, computing an inverse, as well as correcting it, is more expensive than using an LU factorization: for the computation itself, the inverse is more expensive by a constant factor of 3 (assuming classic matrix arithmetic), and for `InverseEC` the complexity of [32, Theorem 8.3] requires the fast selection of linearly independent rows using [7], which might be prohibitive in practice.

For these reasons, we prefer to solve systems using an LU factorization. The goal of the remainder of this section is to do this with a similar complexity as for Algorithm 3 and while avoiding to rely on the sophisticated algorithm from [7].

4.1 Small right-hand side

An intermediate solution, requiring the same amount of extra data as the version with the inverse matrix, but using only fast routines, can be as follows. Use as extra data a candidate factorization $\mathcal{LU}$ and a candidate intermediate right-hand side $\mathcal{Y}$ of dimension $m{\times}n$, such that $\mathcal{Y}, \mathcal{U}$ are approximations to the true Y, U with $YU = B$. We simultaneously correct errors in $\mathcal{X}$, $\mathcal{L}$, $\mathcal{U}$, and $\mathcal{Y}$ as follows:

(1) `CroutEC`$(\mathcal{L}\backslash\mathcal{U}, A, 0, n, \varepsilon/3)$; ▹ L and U with $A = LU$
(2) `URTrsmEC`$(\mathcal{Y}, B, U, m, 0, n, \varepsilon/3)$; ▹ Y with $YU = B$
(3) `LRTrsmEC`$(\mathcal{X}, Y, L, m, 0, n, \varepsilon/3)$. ▹ X with $XL = Y$

Note, of course, that if the number of rows m in B is very small, say only $m \leqslant n^{o(1)}$, then it is faster to recover $\mathcal{L}$ and $\mathcal{U}$ only, by running `CroutEC`, and then compute Y and X directly from the corrected L and U with classical TRSMs.

4.2 Large right-hand side

If the row dimension m of B is large with respect to the column dimension n, then the matrix $\mathcal{Y}$ from above will be larger than $\mathcal{U}$. The client can instead ask the server to provide as extra data $\mathcal{R}$ as a candidate for U^{-1}, to correct it with U, and then to use L and U^{-1} to correct $\mathcal{X}$ directly:

(1) `CroutEC`$(\mathcal{L}\backslash\mathcal{U}, A, 0, n, \varepsilon/3)$; ▹ L and U with $A = LU$
(2) `TrInvEC`$(\mathcal{R}, U, n, \varepsilon/3)$; ▹$R = U^{-1}$
(3) `LRTrsmEC`$(\mathcal{X}, BR, L, m, n, n, \varepsilon/3)$, ▹$XL = BR = BU^{-1}$

with TrInvEC a variant of InverseEC sketched below as Algorithm 4 (where TRSV is a triangular system solve with a column vector and TRMV is a matrix-vector multiplication). This does not require the expensive algorithm of [7] to select independent columns, as the matrix is triangular.

Note that, in the call to LRTrsmEC in the last step, the right-hand side BR is left unevaluated, just as in the calls to TrsmEC from Algorithm 3.

Algorithm 4 TrInvEC($U, \mathcal{R}, n, \varepsilon$) corrects $\mathcal{R} + E = U^{-1}$

```
$P_J$ = ColSupport($U^{-1}v - \mathcal{R}v$);  ▷ $U^{-1}v$ via TRSV with $U$, $\mathcal{R}v$ via TRMV with candidate
$T = (P_J^T U P_J)^{-1}$;  ▷ $O(r^\omega)$
$E' = (EP_J) = (I - \mathcal{R}U)P_J T^{-1}$, so it can be recovered via multi-sparse interpolation as $VE' = \left(VP_J - (V\mathcal{R})(UP_J)\right) T^{-1}$.
```

5 CONCLUSION

We have shown how to efficiently correct errors in an LU factorization, and how to apply this error correction to system solving.

A few remaining challenges are to:

- Generalize our error-correcting algorithms to matrices A which do not have GRP, correcting more general factorizations such as $A = PLUQ$ where P, Q are permutation matrices. Our approach works directly if the permutations P and Q are known to be error-free, but correcting erroneous permutations $\mathcal{P}$ and $\mathcal{Q}$ is more difficult.
- Directly correct errors in $\mathcal{L}$ and $\mathcal{R}$ such that $AR = L$, i.e., $R = U^{-1}$. This would be useful for system solving, as we have seen above.
- More generally, correcting errors only in the solution X of a linear system, without any extra information from the server, would be an even more ambitious goal.

REFERENCES

[1] Carlos Arreche (Ed.). 2018. *ISSAC'2018, New York, USA*. ACM, New York.

[2] M. Ben-Or and P. Tiwari. 1988. A Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM Press, New York, NY, USA, 301–309.

[3] Janko Böhm, Wolfram Decker, Claus Fieker, and Gerhard Pfister. 2015. The use of bad primes in rational reconstruction. *Math. Comput.* 84, 296 (2015), 3013–3027. https://doi.org/10.1090/mcom/2951

[4] Aurelien Bouteiller, Thomas Herault, George Bosilca, Peng Du, and Jack Dongarra. 2015. Algorithm-Based Fault Tolerance for Dense Matrix Factorizations, Multiple Failures and Accuracy. *ACM Trans. Parallel Comput.* 1, 2, Article 10 (Feb. 2015), 28 pages. https://doi.org/10.1145/2686892

[5] Brice Boyer and Erich L. Kaltofen. 2014. Numerical Linear System Solving with Parametric Entries by Error Correction. In *Proceedings of the 2014 Symposium on Symbolic-Numeric Computation (SNC '14)*. ACM, New York, NY, USA, 33–38. https://doi.org/10.1145/2631948.2631956

[6] James R. Bunch and John E. Hopcroft. 1974. Triangular Factorization and Inversion by Fast Matrix Multiplication. *Math. Comp.* 28, 125 (1974), 231–236. https://doi.org/10.1090/S0025-5718-1974-0331751-8

[7] Ho Yee Cheung, Tsz Chiu Kwok, and Lap Chi Lau. 2013. Fast Matrix Rank Algorithms and Applications. *J. ACM* 60, 5 (Oct. 2013), 31:1–31:25. https://doi.org/10.1145/2528404

[8] Matthew T. Comer, Erich L. Kaltofen, and Clément Pernet. 2012. Sparse Polynomial Interpolation and Berlekamp/Massey Algorithms That Correct Outlier Errors in Input Values. In *ISSAC'2012, Grenoble, France*, Joris van der Hoeven and Mark van Hoeij (Eds.). ACM, New York, 138–145. https://doi.org/10.1145/2442829.2442852

[9] Teresa Davies and Zizhong Chen. 2013. Correcting Soft Errors Online in LU Factorization. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing (HPDC '13)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/2493123.2462920

[10] Jack J. Dongarra, Lain S. Duff, Danny C. Sorensen, and Henk A. Vander Vorst. 1998. *Numerical Linear Algebra for High Performance Computers*. SIAM, Philadelphia, PA, USA. https://doi.org/10.1137/1.9780898719611

[11] P. Du, P. Luszczek, and J. Dongarra. 2011. High Performance Dense Linear System Solver with Soft Error Resilience. In *2011 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Washington, D.C., USA, 272–280. https://doi.org/10.1109/CLUSTER.2011.38

[12] Jean-Guillaume Dumas, Thierry Gautier, Clément Pernet, and Ziad Sultan. 2014. Parallel Computation of Echelon Forms. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Springer International Publishing, Cham, 499–510. https://doi.org/10.1007/978-3-319-09873-9_42

[13] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. 2008. Dense Linear Algebra over Prime Fields. *ACM Trans. Math. Software* 35, 3 (Nov. 2008), 1–42. https://doi.org/10.1145/1391989.1391992

[14] Jean-Guillaume Dumas and Erich Kaltofen. 2014. Essentially Optimal Interactive Certificates in Linear Algebra, See [29], 146–153. https://doi.org/10.1145/2608628.2608644

[15] Jean-Guillaume Dumas, Erich Kaltofen, Emmanuel Thomé, and Gilles Villard. 2016. Linear Time Interactive Certificates for the Minimal Polynomial and the Determinant of a Sparse Matrix. In *ISSAC'2016, Waterloo, ON, Canada*, Xiao-Shan Gao (Ed.). ACM, New York, 199–206. https://doi.org/10.1145/2930889.2930908

[16] Jean-Guillaume Dumas, David Lucas, and Clément Pernet. 2017. Certificates for Triangular Equivalence and Rank Profiles, See [34], 133–140. https://doi.org/10.1145/3087604.3087609

[17] Jean-Guillaume Dumas, Clément Pernet, and Ziad Sultan. 2013. Simultaneous Computation of the Row and Column Rank Profiles, See [27], 181–188. https://doi.org/10.1145/2465506.2465517

[18] Rūsiņš Freivalds. 1979. Fast Probabilistic Algorithms. In *Mathematical Foundations of Computer Science 1979 (Lecture Notes in Computer Science)*, J. Bečvář (Ed.), Vol. 74. Springer-Verlag, Olomouc, Czechoslovakia, 57–69. https://doi.org/10.1007/3-540-09526-8_5

[19] François Le Gall. 2014. Powers of Tensors and Fast Matrix Multiplication, See [29], 296–303. https://doi.org/10.1145/2608628.2608664

[20] Leszek Gąsieniec, Christos Levcopoulos, Andrzej Lingas, Rasmus Pagh, and Takeshi Tokuyama. 2017. Efficiently Correcting Matrix Products. *Algorithmica* 79, 2 (01 Oct 2017), 428–443. https://doi.org/10.1007/s00453-016-0202-3

[21] O. Goldreich, D. Ron, and M. Sudan. 2000. Chinese remaindering with errors. *IEEE Transactions on Information Theory* 46, 4 (July 2000), 1330–1338. https://doi.org/10.1109/18.850672

[22] Kuang-Hua Huang and Jacob A. Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. Computers* 33, 6 (1984), 518–528.

[23] E. Kaltofen and L. Yagati. 1988. Improved Sparse Multivariate Polynomial Interpolation Algorithms. In *ISSAC '88*. Springer Verlag, Berlin, Heidelberg, 467–474.

[24] Erich L. Kaltofen, Michael Nehring, and B. David Saunders. 2011. Quadratic-Time Certificates in Linear Algebra. In *ISSAC'2011, San Jose, California, USA*, Anton Leykin (Ed.). ACM, New York, 171–176. https://doi.org/10.1145/1993886.1993915

[25] Erich L. Kaltofen, Clément Pernet, Arne Storjohann, and Cleveland Waddell. 2017. Early Termination in Parametric Linear System Solving and Rational Function Vector Recovery with Error Correction, See [34], 237–244. https://doi.org/10.1145/3087604.3087645

[26] Erich L. Kaltofen and Zhengfeng Yang. 2013. Sparse Multivariate Function Recovery from Values with Noise and Outlier Errors, See [27], 219–226. https://doi.org/10.1145/2465506.2465524

[27] Manuel Kauers (Ed.). 2013. *ISSAC'2013, Boston, USA*. ACM, New York.

[28] Majid Khonji, Clément Pernet, Jean-Louis Roch, Thomas Roche, and Thomas Stalinski. 2010. Output-sensitive Decoding for Redundant Residue Systems. In *ISSAC'2010, Munich, Germany*, Wolfram Koepf (Ed.). ACM, New York, 265–272. https://doi.org/10.1145/1837934.1837985

[29] Katsusuke Nabeshima (Ed.). 2014. *ISSAC'2014, Kobe, Japan*. ACM, New York.

[30] Rasmus Pagh. 2013. Compressed Matrix Multiplication. *ACM Trans. Comput. Theory* 5, 3, Article 9 (Aug. 2013), 17 pages. https://doi.org/10.1145/2493252.2493254

[31] R. Prony. 1795. Essai expérimental et analytique sur les lois de la dilatabilité des fluides élastiques et sur celles de la force expansive de la vapeur de l'eau et de la vapeur de l'alkool, à différentes températures. *J. de l'École Polytechnique Floréal et Plairial, an III* 1, cahier 22 (1795), 24–76.

[32] Daniel S. Roche. 2018. Error Correction in Fast Matrix Multiplication and Inverse, See [1], 343–350. https://doi.org/10.1145/3208976.3209001

[33] D. S. Roche. 2018. What Can (and Can'T) We Do with Sparse Polynomials?, See [1], 25–30.

[34] Mohab Safey El Din (Ed.). 2017. *ISSAC'2017, Kaiserslautern, Germany*. ACM, New York.