

# EXACT SOLUTION OF SPARSE LINEAR SYSTEMS VIA LEFT-LOOKING ROUND-OFF-ERROR-FREE LU FACTORIZATION IN TIME PROPORTIONAL TO ARITHMETIC WORK\*

CHRISTOPHER LOURENÇO<sup>†</sup>, ADOLFO R. ESCOBEDO<sup>‡</sup>, ERICK MORENO-CENTENO<sup>§</sup>,  
AND TIMOTHY A. DAVIS<sup>¶</sup>

**Abstract.** The roundoff-error-free (REF) LU factorization, along with the REF forward and backward substitution algorithms, allows a rational system of linear equations to be solved exactly and efficiently. The REF LU factorization framework has two key properties: all operations are integral, and the size of each entry is bounded polynomially—a bound that rational arithmetic Gaussian elimination achieves only via the use of computationally expensive greatest common divisor operations. This paper develops a sparse version of REF LU, termed the Sparse Left-looking Integer-Preserving (SLIP) LU factorization, which exploits sparsity while maintaining integrality of all operations. In addition, this paper derives a tighter polynomial bound on the size of entries in  $L$  and  $U$  and shows that the time complexity of SLIP LU is proportional to the cost of the arithmetic work performed. Last, SLIP LU is shown to significantly outperform a modern full-precision rational arithmetic LU factorization approach on a set of real world instances. In all, SLIP LU is a framework to efficiently and exactly solve sparse linear systems.

**Key words.** exact linear solutions, LU factorizations, roundoff errors, solving linear systems, sparse matrix algorithms, sparse IPGE word length, sparse linear systems

**AMS subject classifications.** 15A23, 90C05, 65F50, 65F05, 65G50, 15A15

**DOI.** 10.1137/18M1202499

**1. Introduction.** Obtaining the exact rational solution of a *sparse* system of linear equations (SLEs),  $Ax = b$ , is key to many applications within fields such as number theory [16, 45], physics [30], computational geometry [22, 6], and exact linear/integer programming [15, 7]. However, solving these systems exactly is challenging due to the roundoff errors intrinsic to floating point calculations. Individually, roundoff errors are harmless; however, they may propagate, accumulate, and magnify, ultimately rendering the output of linear solvers egregiously incorrect. Roundoff errors are particularly problematic for linear systems which are ill-conditioned, numerically unstable, or require many operations to solve [34]. It is imperative, therefore, that efficient algorithms and robust software exist which account for, and ultimately eliminate, roundoff errors when solving linear systems.

This paper focuses on exactly solving the sparse linear systems which arise in exact linear programming (LP). The focus on solving linear programs (LPs) exactly

\*Received by the editors July 24, 2018; accepted for publication (in revised form) by M. Tuma March 4, 2019; published electronically May 14, 2019.

<http://www.siam.org/journals/simax/40-2/M120249.html>

**Funding:** The work of the authors was partially supported by the National Science Foundation under grant CMMI-1252456. The work of the first author was also supported by the Texas A&M Graduate Merit Fellowship.

<sup>†</sup>Department of Industrial and Systems Engineering, Texas A&M University, College Station, TX 77843 (clouren@tamu.edu).

<sup>‡</sup>School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85281 (adRes@asu.edu).

<sup>§</sup>Department of Industrial and Systems Engineering, Texas A&M University, College Station, TX 77843, and Department of Industrial and Operations Engineering, Instituto Tecnológico Autónomo de México, 01080 Ciudad de México, México (emc@tamu.edu).

<sup>¶</sup>Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843 (davis@tamu.edu).

is motivated by the fact that 72% of the real world LPs within the NETLIB LP repository are ill-conditioned [41], leading to the conjecture that ill-conditioned LPs are frequent in practice [40]. Indeed, methods to solve LPs exactly have been researched because commercial LP solvers come with limited guarantees: they may classify sub-optimal solutions as optimal, or even feasible problems as infeasible and vice versa [15, 36]. Therefore, commercial solvers are inadequate for those applications which require exact solutions, such as feasibility problems, compiler optimization, and generating mixed integer rounding cuts in integer programming [21, 43].

There are currently two approaches for solving LPs exactly: certify and repair [15, 3] and LP iterative refinement [27, 26]. Each method employs a mixed precision approach in which the majority of operations are performed in floating point precision. Both guarantee exactness by solving—via full-precision rational arithmetic LU factorizations<sup>1</sup>—the sparse linear systems associated with promising basis matrices. However, these rational LU factorizations may exhibit severe computational slowdown on real world instances. Indeed, rational LU factorizations often represent the bottleneck of exact LP algorithms, in some cases occupying more than 90% of the exact solver’s run time [26]. Despite their prevalent use, very little research has been devoted to improving exact LU factorizations. We note that, aside from rational-arithmetic LU factorization, there exist other methods to solve linear systems exactly, including Wiedemann’s black box [33], p-adic lifting [16, 44], and iterative refinement [44]. These methods were shown to have advantages, particularly when solving linear systems in which the encoding length of the solution is low [8]. Conversely, in the context of exact LP, rational LU factorization typically outperforms these other methods for solutions whose computations require high bit lengths. For this reason, rational LU factorization is utilized in commercial exact LP solvers (QSOpt [2] and Soplex [47, 26]); thus, this paper focuses solely on LU factorization.

To address the drawbacks of rational LU factorizations, the roundoff-error-free (REF) LU factorization was developed. Based on integer-preserving Gaussian elimination (IPGE), the REF LU factorization exactly solves SLEs exclusively in integer arithmetic, thereby avoiding greatest common divisor operations. REF LU has the key property that each entry’s bit length (i.e., the number of bits required for storage) is bounded polynomially [18]. Associated computational tests showed that REF LU decisively outperformed rational LU for exactly solving dense linear systems [20]. Moreover, dense LU update algorithms were derived for REF LU, further accelerating this framework when compared to rational factorization [19].

Despite excellent results for dense matrices, the REF LU framework is not currently equipped to handle the large-scale sparse systems prevalent in modern applications. This paper expands REF LU to the sparse case by developing the theoretical and computational framework for the Sparse Left-looking Integer-Preserving (SLIP) LU factorization. Specifically, this paper makes the following contributions: First, it derives a left-looking approach to obtain the REF LU factorization. Second, it implements sparsity exploiting features into left-looking REF LU to derive SLIP LU. Third, it improves the traditional IPGE maximum bit length for both sparse and symmetric positive definite matrices. Fourth, it shows that the computational complexity of SLIP LU is proportional to the cost of its arithmetic operations. Finally, it computationally shows that SLIP LU outperforms a state-of-the-art, rational arithmetic sparse LU factorization routine.

<sup>1</sup>An LU factorization factors the matrix  $A$  into the product of lower,  $L$ , and upper,  $U$ , triangular matrices; that is,  $A = LU$ .

This paper is organized as follows: section 2 gives an overview of the background required for this paper. Section 3 derives left-looking REF LU and SLIP LU. Section 4 derives the improved IPGE bit length and the computational complexity of SLIP LU. Section 5 presents a computational comparison of SLIP LU to a state-of-the-art rational arithmetic LU factorization. Last, section 6 draws conclusions.

**2. Preliminaries.** This section presents the theoretical foundation on which our algorithms are built; it is organized as follows: Subsection 2.1 gives a brief overview of traditional and integer-preserving Gaussian elimination. Subsection 2.2 discusses an algorithmic extension to IPGE which takes advantage of zeros in sparse matrices. Finally, subsections 2.3 and 2.4 review left-looking LU factorization and the (dense) REF LU factorization, respectively. Note that the ensuing sections assume no row or column permutations are applied to the matrix  $A$ , an assumption that is relaxed beginning in section 5.

**2.1. Traditional and integer-preserving Gaussian elimination.** Gaussian elimination (GE) and IPGE are elimination processes for solving SLEs  $A\mathbf{x} = \mathbf{b}$ . Given a full rank  $n \times n$  matrix  $A$  and right-hand side vector  $\mathbf{b}$ , denote the  $k$ th iteration GE matrix as  $\tilde{A}^k$  and the  $k$ th iteration IPGE matrix as  $A^k$  for  $0 \leq k \leq n$  (with  $\tilde{A}^0 = A^0 = A$ ). Let  $\tilde{a}_{i,j}^k$  and  $a_{i,j}^k$  denote the individual entries of  $\tilde{A}^k$  and  $A^k$  for  $1 \leq i \leq n$ ,  $1 \leq j \leq n$ , and  $0 \leq k \leq n$ , respectively. Finally, denote the  $k$ th pivot element, for  $0 \leq k \leq n$ , of  $\tilde{A}^k$  and  $A^k$  as  $\tilde{\rho}^k$  and  $\rho^k$  (with  $\tilde{\rho}^0 = \rho^0 = 1$ ), respectively. At iteration  $k$ , GE computes the entries  $\tilde{a}_{i,j}^k$  as follows:

$$(1) \quad \tilde{a}_{i,j}^k = \begin{cases} \tilde{a}_{i,j}^{k-1} & \text{if } i = k, \\ \tilde{a}_{i,j}^{k-1} - \frac{\tilde{a}_{k,j}^{k-1} \tilde{a}_{i,k}^{k-1}}{\tilde{\rho}^k} & \text{otherwise.} \end{cases}$$

If the input matrix  $A$  and right-hand side vector  $\mathbf{b}$  are integral, that is,  $A \in \mathbb{Z}^{n \times n}$  and  $\mathbf{b} \in \mathbb{Z}^n$ , then the IPGE algorithm can be applied. Note that if  $A$  or  $\mathbf{b}$  is rational or decimal, it can be made integral by multiplying by the least common multiple or the appropriate power of 10, respectively. Then, at iteration  $k$ , IPGE computes the entries  $a_{i,j}^k$  as

$$(2) \quad a_{i,j}^k = \begin{cases} a_{i,j}^{k-1} & \text{if } i = k, \\ \frac{\rho^k a_{i,j}^{k-1} - a_{k,j}^{k-1} a_{i,k}^{k-1}}{\rho^{k-1}} & \text{otherwise.} \end{cases}$$

Henceforth, we refer to (2) as an “IPGE update.” Notice that (1) becomes  $(\tilde{\rho}^k \tilde{a}_{i,j}^{k-1} - \tilde{a}_{k,j}^{k-1} \tilde{a}_{i,k}^{k-1}) / \tilde{\rho}^k$  by generating a common denominator. With this change, one could see that (1) and (2) have the same structure, differing by only the denominator.

On one hand, this difference is minor as, at iteration  $k$ , both algorithms make the nondiagonal entries of the  $k$ th column of  $A^k$  equal to zero. If the GE or IPGE algorithms are applied in full to the augmented matrix  $[A|\mathbf{b}]$ , each algorithm yields a diagonal matrix augmented by a modified  $\mathbf{b}$ , denoted  $\tilde{\mathbf{b}}'$  and  $\mathbf{b}'$ , respectively. In GE, this diagonal matrix is the identity matrix,  $I$ ; thus the solution to the system  $A\mathbf{x} = \mathbf{b}$  is given as  $\mathbf{x} = \tilde{\mathbf{b}}'$ . Conversely, in IPGE, this diagonal matrix is the identity matrix scaled by the determinant of  $A$ ,  $\det(A)I$ ; thus, the solution to the system is given as  $\mathbf{x} = \mathbf{b}' / \det(A)$ . If GE/IPGE are applied to only rows/columns  $k+1, \dots, n$  of  $A^k$ , then the result of the algorithm is an upper triangular matrix augmented by a

modified  $\mathbf{b}$ ; thus, the solution vector  $\mathbf{x}$  can be computed via a backward substitution algorithm.

On the other hand, this minor difference between GE and IPGE yields a special property, as this modification to GE ensures that every intermediate operation performed in IPGE is integral [17, 4, 39]. This means that every IPGE matrix,  $A^k$ , for  $k = 1, \dots, n$  is comprised exclusively of integral entries. IPGE has three key properties summarized below.

LEMMA 2.1. *The divisions in (2) are integral [17, 4, 39].*

LEMMA 2.2. *Each IPGE entry,  $a_{i,j}^k$ , is a subdeterminant of  $A$  [17].*

LEMMA 2.3. *Let  $\sigma = \max_{i,j} |a_{i,j}^0|$ . The maximum bit length required to store any IPGE entry, denoted  $\beta_{\max}$ , is upper-bounded polynomially as follows [5]:*

$$\beta_{\max} \leq \lceil n \log(\sigma \sqrt{n}) \rceil.$$

For a contemporary in-depth discussion of the key properties of IPGE, we refer the reader to [18].

**2.2. Exploiting zeros in IPGE.** One way in which GE is made more efficient on sparse matrices is by skipping operations on zeros. In particular, in (1), if at iteration  $k$ , either  $\tilde{a}_{k,j}^{k-1} = 0$  or  $\tilde{a}_{i,k}^{k-1} = 0$ , then the corresponding update to  $\tilde{a}_{i,j}^{k-1}$  can be skipped because  $\tilde{a}_{i,j}^k = \tilde{a}_{i,j}^{k-1}$ .

Conversely, in IPGE, if either  $a_{k,j}^{k-1} = 0$  or  $a_{i,k}^{k-1} = 0$ , (2) becomes  $a_{i,j}^k = \rho^k a_{i,j}^{k-1} / \rho^{k-1}$ . Thus, the IPGE update reduces to a multiplication and division by consecutive pivots; a property which Lee and Saunders showed can be exploited [37]. Specifically, suppose a sequence,  $\{l, \dots, m\}$ , of IPGE iterations exists such that for each  $k \in \{l, \dots, m\}$ , either  $a_{i,k}^{k-1} = 0$  or  $a_{k,j}^{k-1} = 0$ . Then,  $a_{i,j}^m$  is computed as

$$a_{i,j}^m = \frac{\rho^m}{\rho^{m-1}} \frac{\rho^{m-1}}{\rho^{m-2}} \cdots \frac{\rho^{l+1}}{\rho^l} \frac{\rho^l}{\rho^{l-1}} a_{i,j}^l.$$

It is clear that the above sequence contains unnecessary operations which can be skipped and condensed into a single update as follows [37]:

$$(3) \quad a_{i,j}^m = \frac{\rho^m a_{i,j}^l}{\rho^{l-1}}.$$

As explained in the next algorithm, the above property can be used to skip these unnecessary operations by using a so-called history matrix which keeps track of the last iteration in which an entry was IPGE updated. This algorithm proceeds as follows. Initialize the history matrix,  $H^k$ , as  $h_{i,j}^0 = 0 \forall i, j$ . Then, for each IPGE iteration  $k = 1, \dots, n$ , if  $a_{i,k}^{k-1} = 0$  or  $a_{k,j}^{k-1} = 0$ , skip the IPGE update on  $a_{i,j}^{k-1}$  and set  $a_{i,j}^k = a_{i,j}^{k-1}$  and  $h_{i,j}^k = h_{i,j}^{k-1}$ . Otherwise,  $a_{i,j}^{k-1}$  must be updated, but, prior to performing this update, since some operations may have been skipped, ensure that  $a_{i,j}^{k-1}$  is at its true value with respect to iteration  $k-1$  via

$$(4) \quad a_{i,j}^{k-1} = \frac{\rho^{k-1} a_{i,j}^{k-2}}{\rho^{h_{i,j}^{k-1}}}.$$

After applying (4), hereafter referred to as a “History update,” apply the IPGE update to  $a_{i,j}^{k-1}$  and set  $h_{i,j}^k = k$ . Notice that this description assumes that at iteration  $k$ , every entry in row  $k$  and column  $k$  is at its true value with respect to iteration  $k - 1$ .

Computationally, Lee and Saunders tested this algorithm versus traditional IPGE on randomly generated matrices with 50%, 20%, and 10% nonzero entries, and saw speed-ups in run time of magnitude 1.77, 1.31, and 3, respectively [37]. Remarkably, these results were obtained without the use of any sparse algorithms or data structures (i.e., all matrices were stored and operated on as fully dense).

**2.3. Left-looking LU (LLU).** Gilbert and Peierls’ left-looking LU factorization constructs the sparse LU factorization  $A = LU$  one column at a time. At iteration  $k$ , LLU accesses the  $k$ th column of  $A$  (denoted  $A(:, k)$ ) and columns  $1 : k - 1$  of  $L$ . Prior to reviewing this factorization we introduce the following notation used throughout the paper.

**DEFINITION 2.4.** Let  $L^k$  denote the  $k$ th left-looking  $L$  matrix for  $k = 0, \dots, n$ . Specifically,  $L^k$  is the  $n \times k$  completed portion of  $L$  augmented by the last  $n - k$  columns of the  $n \times n$  identity matrix.

The following theorem presents the LLU factorization of a matrix  $A$ .

**THEOREM 2.5** (left-looking LU [25]). For a fixed column  $k$ ,  $k = 1, \dots, n$ , the solution to the following lower triangular linear system yields the  $k$ th column of  $L$  and  $U$ :

$$L^{k-1} \mathbf{x} = A(:, k),$$

where  $x_1, \dots, x_k$  are the nonzeros in  $U(:, k)$  and  $x_{k+1}, \dots, x_n$  are the off diagonal nonzeros in  $L(:, k)$  (with  $l_{k,k} = 1$ ).

Gilbert and Peierls showed that the LLU factorization computes  $L$  and  $U$  in time proportional to only the number of necessary floating point operations, subject to the caveat that every sparse LU factorization must use heuristics to solve the NP-Hard problem of finding a fill-reducing ordering to minimize the work in the ensuing factorization [25]. LLU factorization is unique in that it is the only sparse LU factorization algorithm in existence today with this property [25, 14].

**2.4. REF LU factorization.** As mentioned in the introduction, the REF LU factorization, developed by Escobedo and Moreno-Centeno, solves dense SLEs exclusively in integer arithmetic [18]. Formally, the nonsingular matrix  $A \in \mathbb{Z}^{n \times n}$  is factored as  $A = LDU$ , explicitly:

$$(5) \quad A = \begin{bmatrix} \rho^1 & 0 & 0 & \dots & 0 \\ l_{2,1} & \rho^2 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & l_{n,3} & \dots & \rho^n \end{bmatrix} D \begin{bmatrix} \rho^1 & u_{1,2} & u_{1,3} & \dots & u_{1,n} \\ 0 & \rho^2 & u_{2,3} & \dots & u_{2,n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \rho^n \end{bmatrix},$$

where  $D = \text{diag}(\rho^0 \rho^1, \rho^1 \rho^2, \dots, \rho^{n-1} \rho^n)^{-1}$ .

The integrality of entries in  $L$  and  $U$  is attested by the fact that factorization (5) can be rewritten in terms of IPGE entries as [18]

$$(6) \quad A = \begin{bmatrix} a_{1,1}^0 & 0 & 0 & \cdots & 0 \\ a_{2,1}^0 & a_{2,2}^1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ a_{n,1}^0 & a_{n,2}^1 & a_{n,3}^2 & \cdots & a_{n,n}^{n-1} \end{bmatrix} D \begin{bmatrix} a_{1,1}^0 & a_{1,2}^0 & a_{1,3}^0 & \cdots & a_{1,n}^0 \\ 0 & a_{2,2}^1 & a_{2,3}^1 & \cdots & a_{2,n}^1 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n,n}^{n-1} \end{bmatrix}.$$

For randomly generated dense matrices, the REF LU factorization was up to 22 and 17 times faster than rational Crout and Doolittle factorizations, respectively [20].

### 3. Sparse left-looking integer-preserving LU factorization (SLIP LU).

This section derives the theoretical foundation of SLIP LU, which also serves as the first fully sparse implementation of IPGE in both data structures and algorithms. The contents of this section and its organization are as follows. Subsection 3.1 derives a new version of the REF LU factorization, namely the left-looking (dense) REF LU. Subsection 3.2 modifies REF forward substitution [18] in order to exactly solve the lower triangular linear systems required for the left-looking REF LU factorization. Subsection 3.3 develops Algorithm 1, a new sparse REF triangular solve algorithm which forms the basis of SLIP LU, by combining Lee and Saunders' history matrix algorithm [37], our new (dense) REF triangular solve algorithm, and the sparse (floating point) LLU sequence of operations [25]. Finally, subsection 3.4 describes how to use the SLIP LU factorization to solve a sparse linear system  $A\mathbf{x} = \mathbf{b}$ .

**3.1. Left-looking REF LU.** This subsection derives the left-looking REF LU factorization of the matrix  $A \in \mathbb{Z}^{n \times n}$ . At iteration  $k$ , left-looking REF LU accesses the first  $k-1$  completed columns of  $L$  and  $D$  and computes the  $k$ th column of  $L$  and  $U$ . Theorem 3.1 formally presents the factorization.

**THEOREM 3.1.** *The REF LU factorization, given by (5) and (6), can be obtained in a left-looking fashion by solving partial lower triangular linear systems of the form  $L^{k-1}D^{k-1}\mathbf{x} = A(:,k)$  for  $k = 1, \dots, n$ :*

$$(7) \quad L^{k-1}D^{k-1}\mathbf{x} = \left[ \begin{array}{cccc|c} \frac{1}{\rho^0} & 0 & 0 & 0 & \mathbf{0} \\ \frac{l_{2,1}}{\rho^1} & \frac{1}{\rho^1} & 0 & 0 & \\ \vdots & & \ddots & & \\ \frac{l_{k-1,1}}{\rho^1} & \cdots & \cdots & \frac{1}{\rho^{k-2}} & \\ \hline \frac{l_{k,1}}{\rho^1} & \cdots & \cdots & \frac{l_{k,k-1}}{\rho^{k-2}\rho^{k-1}} & \\ \vdots & & & \vdots & \\ \frac{l_{n,1}}{\rho^1} & \cdots & \cdots & \frac{l_{n,k-1}}{\rho^{k-2}\rho^{k-1}} & \frac{1}{\rho^{k-1}}\mathbf{I} \end{array} \right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{1,k}^0 \\ a_{2,k}^0 \\ \vdots \\ a_{n,k}^0 \end{bmatrix},$$

where  $L^k$  is defined as in subsection 2.3 and  $D^k$  is the first completed  $k$  columns of  $D$  augmented by the last  $n-k$  columns of  $(1/\rho^k)I$ .

The proof of Theorem 3.1 relies on the following lemma.

**LEMMA 3.2.** *For any  $k \in \{1, \dots, n\}$ , assume the partial matrix  $L^{k-1}D^{k-1}$  is correct. Then, solving the lower triangular system given by (7) yields the following*

elements of the REF LU factorization:

$$x_j = \begin{cases} u_{j,k} & \text{for } j = 1, \dots, k-1, \\ \rho^k = u_{k,k} = l_{k,k} & \text{for } j = k, \\ l_{j,k} & \text{for } j = k+1, \dots, n. \end{cases}$$

The proofs of Lemma 3.2 and Theorem 3.1 are relegated to Appendices A and B, respectively.

The key step of the left-looking REF LU factorization is to solve the partial linear system  $L^{k-1}D^{k-1}\mathbf{x} = A(:, k)$  for any column  $k$ . Since (7) is a lower triangular system, solving it via forward substitution yields

$$(8) \quad x_j = \rho^{\min(j,k)-1} \left( a_{j,k}^0 - \sum_{i=1}^{\min(j,k)-1} \frac{l_{j,i}x_i}{\rho^{i-1}\rho^i} \right) \quad \text{for } j = 1, \dots, n.$$

Assuming unlimited precision, (8) will give the correct (integral) value of each  $x_j$ . However, directly using this equation to compute each  $x_j$  contains intermediate divisions which are inexact, thereby leading to roundoff errors (specifically the terms  $\frac{l_{j,i}x_i}{\rho^{i-1}\rho^i}$  are, in general, not integral). Therefore, the next subsection shows how to solve this system without roundoff errors.

**3.2. Dense REF triangular solve.** For any column  $k$ ,  $k = 1, \dots, n$ , Theorem 3.3 gives the dense REF triangular solve algorithm.

**THEOREM 3.3.** *The system  $L^{k-1}D^{k-1}\mathbf{x} = A(:, k)$  can be solved without roundoff error by the following algorithm: Initialize  $\mathbf{x} = A(:, k)$ . Then for each  $j = 2, \dots, n$ , apply (9),*

$$(9) \quad x_j = \frac{\rho^j x_j - l_{j,i} x_i}{\rho^{i-1}} \quad \text{for iteration } i = 1, \dots, \min(j, k) - 1.$$

Theorem 3.3 is a specialized version of the REF forward substitution from [18]. In fact, the major difference is that for each  $x_j$ , REF forward substitution performs operations until iteration  $j-1$ . However, (9) terminates at iteration  $\min(j, k)-1$  due to the special structure of  $L^{k-1}D^{k-1}$ . Therefore, the proof of Theorem 3.3 is omitted because it is essentially the same as the proof of REF forward substitution in [18].

**3.3. Sparse REF triangular solve.** This section derives Algorithm 1, an efficient sparse REF triangular solve algorithm, the key theoretical result of this paper. Since the SLIP LU factorization consists of  $n$  iterations of this triangular solve, this section also serves to formally derive the factorization. To derive this efficient triangular solve algorithm, sparsity exploiting features are integrated into the algorithm described in Theorem 3.3 so that the partial linear system  $L^{k-1}D^{k-1}\mathbf{x} = A(:, k)$  can be solved efficiently. As with many traditional sparse algorithms, our approach comprises both a symbolic and a numeric phase.

**3.3.1. Sparse REF triangular solve: Symbolic phase.** The first step of solving the  $k$ th triangular system  $L^{k-1}D^{k-1}\mathbf{x} = A(:, k)$  is to compute the nonzero pattern of  $\mathbf{x}$ , denoted  $\mathcal{X}$ , in order to expedite the computation of the numeric values of  $\mathbf{x}$ . From (9), for any  $j = 1, \dots, n$ , there are two cases in which  $x_j$  can be nonzero.

Case 1:  $a_{j,k}^0 \neq 0$ . In this case,  $x_j$  is initialized to a nonzero value. Thus, ignoring any potential numeric cancellation, the final  $x_j$  will also be nonzero.

Case 2:  $a_{j,k}^0 = 0$ , but there is at least one  $i < j$  such that  $l_{j,i} \neq 0$  and  $x_i \neq 0$ . In this case,  $x_j$  is initialized to zero, but the sequence of  $\min(j, k) - 1$  iterative updates will add at least one nonzero term to  $x_j$ . As a result, ignoring any potential numeric cancellation, the final  $x_j$  will also be nonzero.

These two cases for SLIP LU are identical to those for LLU [25]; therefore, the traditional algorithm to obtain  $\mathcal{X}$  can be used. This algorithm performs a sequence of depth first searches (DFS) on a graph,  $G_L$ , of  $n$  nodes, where a directed edge  $(i, j)$  exists if  $l_{j,i} \neq 0$ . The nonzero pattern  $\mathcal{X}$  is given as the indices of the set of nodes reachable, via DFS, from each node  $j$  in which  $a_{j,k}^0 \neq 0$ . Consequently, this algorithm is referred to as obtaining the reach of the  $k$ th column of  $A$  on the graph of  $L$  and is denoted  $\mathcal{X} = \text{Reach}_{G_L}(A(:, k))$ .

As a result of  $\mathcal{X}$  being obtained via a sequence of DFS, the output of this algorithm is the set of nonzero indices in topological order. Unlike in LLU, where operations can be performed in topological order [25], due to the special structure of (9), the sparse REF triangular solve requires that the vector  $\mathcal{X}$  be sorted prior to numeric computation. We show in subsection 4.2.1 that, in spite of this sort, our algorithm still requires work proportional to the cost of its arithmetic work.

**3.3.2. Sparse REF triangular solve: Numeric computation.** Given the nonzero pattern  $\mathcal{X}$ , the next step of the sparse REF triangular solve is to compute the numeric values of  $\mathbf{x}$ . Below, we present two algorithms to compute  $\mathbf{x}$ , both of which traverse  $\mathcal{X}$  in its sorted order. The first, described in the following two paragraphs, is a simple, yet inefficient, algorithm whose purpose is to aid in understanding and to illustrate the correctness of the sparse REF triangular solve which follows.

Given a nonzero  $x_j$ , initialize it as  $x_j = a_{j,k}^0$ . To determine which operations must be performed on  $x_j$ , as per (9), we define a set  $\mathcal{Y}$  comprising the indices of the nonzero  $x_i$  and  $l_{j,i}$  whose product must be added to  $x_j$ . That is,  $\mathcal{Y} = \{i : i \in \mathcal{X}, i < j, l_{j,i} \neq 0\}$ ; for any  $i \in \mathcal{Y}$ , we say that  $x_j$  depends on  $x_i$ . Notice that  $\mathcal{Y}$  can be an empty set (such as when  $x_j$  is the first nonzero in  $\mathcal{X}$ ).

The set  $\mathcal{Y}$  may contain nonconsecutive indices; therefore, much like in subsection 2.2, intermediate IPGE operations need to be skipped for efficiency. To do this, a history value,  $h_j$ , is initialized as  $h_j = 0$  and used to keep track of the sequence of operations performed on  $x_j$ . Then, for each  $i \in \mathcal{Y}$ , the following sequence of operations are performed. First, the History update is applied, where  $x_j$  is multiplied by the term  $\rho^{i-1}/\rho^{h_j}$ , thus  $x_j = a_{j,k}^{i-1}$ . Next, the IPGE update (via (9)) is applied and we set  $h_j = i$ . Once these two operations have been performed for each  $i \in \mathcal{Y}$ ,  $x_j$  is brought to its final value via a *terminal* History update as follows. If  $x_j \in U(:, k)$ , it is multiplied by  $\rho^{j-1}/\rho^{h_j}$ , thereby obtaining its final value with respect to IPGE iteration  $j - 1$ . Otherwise,  $x_j \in L(:, k)$ ; thus it is multiplied by  $\rho^{k-1}/\rho^{h_j}$  and obtains its final value with respect to IPGE iteration  $k - 1$ .

The above algorithm, though correct, is inefficient for two major reasons: (1) it requires irregular memory access by using previously computed elements in  $\mathbf{x}$ , and (2) it iterates across both  $\mathcal{X}$  and row  $j$  of  $L$  for each nonzero  $x_j$  (to obtain  $\mathcal{Y}$ ). The efficient sparse REF triangular solve, Algorithm 1 below, remedies these inefficiencies by solving the system in a left-looking fashion as follows. First, to avoid irregular memory access, when an  $x_j$  is accessed, it is History updated and finalized, thereby obtaining its correct value with respect to IPGE iteration  $j - 1$  or  $k - 1$  if  $x_j \in U(:, k)$  or  $x_j \in L(:, k)$ , respectively. Then, each  $x_i \in \mathcal{X}$  such that  $i > j$  and  $l_{i,j}$  is nonzero (i.e.,

each  $x_i$  which depends on  $x_j$ ) is IPGE updated via (9). Additionally, a global history vector  $\mathbf{h}$  is utilized, where, at each iteration,  $\mathbf{h}$  is initialized as  $h_j = 0$  only for  $j \in \mathcal{X}$ . With the appropriate data structures, this algorithm performs only the necessary operations and does not require checking if entries are nonzero. Therefore, Algorithm 1 enables the efficient and exact solution of the linear system  $L^{k-1}D^{k-1}\mathbf{x} = A(:, k)$ .

---

**Algorithm 1** Sparse REF Triangular Solve.
 

---

```

1:  $\mathbf{x} = A(:, k)$ 
2:  $\mathcal{X} = \text{Reach}_{G_L}(A(:, k))$ 
3: sort( $\mathcal{X}$ )
4: Reset history vector:  $h_j = 0 \ \forall j \in \mathcal{X}$ 
5: for  $j \in \mathcal{X}$  do
6:   if  $h_j < \min(j, k) - 1$  then
7:     History update:  $x_j = \frac{x_j \rho^{\min(j, k) - 1}}{\rho^{h_j}}$ 
8:   end if
9:   for  $i > j$  and  $l_{i,j} \neq 0$  do
10:    if  $h_i < j - 1$  then
11:      History update:  $x_i = \frac{x_i \rho^{j-1}}{\rho^{h_i}}$ 
12:    end if
13:    IPGE update:  $x_i = \frac{\rho^j x_i - l_{i,j} x_j}{\rho^{j-1}}$ 
14:    History Vector Update:  $h_i = j$ 
15:  end for
16: end for

```

---

**3.4. Using SLIP LU to solve  $A\mathbf{x} = \mathbf{b}$ .** The steps required to solve a sparse linear system via the SLIP LU factorization are similar to those required by REF LU in the dense case [18]; thus we only review them briefly here. The core of the algorithm is to compute the factorization  $A = LDU$ , via  $n$  applications of the sparse REF triangular solve given by Algorithm 1. After this factorization is computed, it is used to solve two triangular linear systems  $LD\mathbf{y} = \mathbf{b}$  and  $U\mathbf{x} = \mathbf{y}$ . The lower triangular system  $LD\mathbf{y} = \mathbf{b}$  is solved via Algorithm 1, with  $\mathcal{X} = \text{Reach}_{G_A}(\mathbf{b})$  if  $\mathbf{b}$  is sparse [23], or  $\mathcal{X} = \{1, \dots, n\}$  otherwise. The upper triangular system  $U\mathbf{x} = \mathbf{y}$  cannot be directly solved in integer arithmetic [18]; therefore, it is scaled by the determinant of  $A$ ,  $\rho^n$  [17]; specifically, we solve the system  $U\mathbf{x} = \rho^n \mathbf{y}$ . This scaling ensures that all intermediate operations, and the final solution vector,  $\mathbf{x}$ , are integral; therefore, this system can be solved with any traditional upper triangular solve algorithm without roundoff error. Finally, the solution to the system  $A\mathbf{x} = \mathbf{b}$  is obtained as  $\mathbf{x} = \mathbf{x}/\rho^n$ . Notice that every step prior to the final computation was done in integer arithmetic; therefore, the final (rational) solution can be expressed exactly, or to any desired level of precision.

**4. Complexity analysis.** This section derives a tighter upper bound on the maximum size of IPGE entries for both sparse and symmetric positive definite (SPD) matrices. Using these new bounds, this section derives the computational complexity of the sparse REF triangular solve and the SLIP LU factorization. Last, it draws some parallels between the computational complexities of the SLIP LU and the LLU factorization.

**4.1. Updated IPGE maximum bit length.** As discussed in subsection 2.1, since IPGE entries are subdeterminants of the input matrix  $A$ , the worst-case maximum bit length of entries, denoted  $\beta_{max}$ , depends on the dimension of the input matrix and the largest initial entry in  $A$  (denoted  $\sigma$ ). Using Hadamard's bound on the determinant size [29, 31], the bit length of each IPGE entry—hence each entry in the REF LU and SLIP LU factorization—is bounded above by [17, 18]:

$$\beta_{max} \leq \lceil n \log(\sigma \sqrt{n}) \rceil.$$

As shown below, this bound is overly pessimistic for many practical problems; indeed, subsections 4.1.1 and 4.1.2 present a tighter bound on  $\beta_{max}$  for sparse and SPD matrices, respectively.

**4.1.1. IPGE bit length for sparse matrices.** Hadamard's bound is tight if and only if the rows or columns of the input matrix  $A$  are orthogonal; otherwise it is overly pessimistic [31]. Moreover, since this bound applies to dense matrices, it is expected to be especially poor for sparse matrices. The new bound relies on the sparsity of  $A$ ; therefore, we define the following term.

**DEFINITION 4.1.** *Let  $\gamma$  be the smallest of two integers: the number of nonzeros in the most dense row of  $A$ , or the number of nonzeros in the most dense column of  $A$ .*

Theorem 4.2 tightens  $\beta_{max}$  for IPGE on sparse matrices.

**THEOREM 4.2.** *In a sparse matrix, the maximum bit length of any IPGE entry is upper-bounded as*

$$(10) \quad \beta_{max} \leq \lceil n \log(\sigma \sqrt{\gamma}) \rceil.$$

The proof of Theorem 4.2 is relegated to Appendix C. Since  $\gamma \leq n$ , the bound presented in Theorem 4.2 is only equivalent to the general IPGE worst-case bit length if  $A$  has both a fully dense row and column. However, even in this case, Theorem 4.2 holds for any sparse submatrix of  $A$ .

**4.1.2. IPGE bit length for SPD matrices.** Theorem 4.3 tightens  $\beta_{max}$  for an SPD matrix.

**THEOREM 4.3.** *In a SPD matrix, the maximum bit length of any IPGE entry is upper-bounded as*

$$(11) \quad \beta_{max} \leq \lceil n \log \sigma \rceil.$$

The proof of Theorem 4.3 is relegated to Appendix D. Though the above bound is tighter, it is still generally pessimistic, as Hadamard's bound is tight for an SPD  $A$  if and only if  $A$  is diagonal. However, we can apply Theorem 4.3 to improve the worst-case complexity of the REF Cholesky factorization. Specifically, in [18], this factorization's complexity was given as

$$\mathcal{O}(n^3(n \log \sigma \sqrt{n} \log(n \log \sigma \sqrt{n}) \log \log(n \log \sigma \sqrt{n}))).$$

Using inequality (11), this complexity reduces to

$$\mathcal{O}(n^3(n \log \sigma \log(n \log \sigma) \log \log(n \log \sigma))).$$

**4.2. Computational complexity.** Subsections 4.2.1 and 4.2.2 give the computational complexity of the REF sparse triangular solve and the SLIP LU factorization, respectively. The derivation of these complexities requires the following concepts.

LEMMA 4.4. *Based on the best-known fast Fourier transform algorithm, the cost of performing multiplications and divisions on two integers of bit length  $\beta$  is given by  $\mathcal{O}(\beta \log \beta \log \log \beta)$  [35, 42].*

DEFINITION 4.5. *For a fixed column  $k$ , let  $\tilde{f}$  denote the total number of arithmetic operations required in LLU to solve the linear system  $L^{k-1}\mathbf{x} = A(:, k)$ . Likewise, let  $f$  denote the total number of arithmetic operations required to compute the full LLU factorization.*

DEFINITION 4.6. *For a fixed column  $k$ , let  $\tilde{I}$  denote the total number of arithmetic operations required in SLIP LU to solve the linear system  $L^{k-1}D^{k-1}\mathbf{x} = A(:, k)$ . Likewise, let  $I$  denote the total number of arithmetic operations required to compute the full SLIP LU factorization.*

Lemma 4.4 is necessary due to the fact that, for exact integer based algorithms, the complexity analysis must include the cost of performing operations on arbitrarily large integers. Definition 4.5 relates to the computational complexity of LLU. Gilbert and Peierls proved that the complexity of LLU's triangular solve depends only on the number of operations performed ( $\tilde{f}$ ), not the matrix dimension [25]. Since the overall LLU factorization requires performing the triangular solve for each column of the input matrix, its overall complexity is also dependent only on  $f$ . Using this same idea, we will use Definition 4.6 to define the computational complexity of SLIP LU.

**4.2.1. REF sparse triangular solve complexity.** This section shows that the complexity of the REF sparse triangular solve is proportional to its arithmetic work. Specifically, Theorem 4.7 formally presents the complexity of the triangular solve.

THEOREM 4.7. *The computational complexity of the REF sparse triangular solve, Algorithm 1, is  $\mathcal{O}(\tilde{I}[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}])$ .*

*Proof.* The complexity presented in Theorem 4.7 can be obtained by analyzing Algorithm 1. Lines 1, 3, and 4 have complexities  $\mathcal{O}(|A(:, k)|)$ ,  $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|)$ , and  $\mathcal{O}(|\mathcal{X}|)$ , respectively. For line 2, Gilbert and Peierls showed that the cost to obtain  $\mathcal{X}$  takes time proportional to the number of traversed edges in the graph of  $L$  [25]. This number of traversed edges is upper-bounded by the integer operations performed  $\mathcal{O}(\tilde{I})$  due to the fact that each edge traversed corresponds to a dependency in the triangular solve (lines 9–15 in Algorithm 1); thus the complexity of line 2 is  $\mathcal{O}(\tilde{I})$ .

Next, we analyze lines 5–16 of Algorithm 1. These lines correspond to computing the numeric values of  $\mathbf{x}$  and require  $\mathcal{O}(\tilde{I}[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}])$  time. This cost follows from comparing GE and IPGE. Gilbert and Peierls showed that, since data access and arithmetic operations are performed only on nonzero entries, their triangular solve requires  $\mathcal{O}(\tilde{f})$  operations [25]. Using the History update, Lee and Saunders showed that unnecessary IPGE operations (i.e., operations on zeros) can be skipped [37]. Therefore, the History update, combined with sparse data structures and the nonzero pattern  $\mathcal{X}$ , allows Algorithm 1 to access and perform operations exclusively on nonzero entries. Thus, using the same logic as Gilbert and Peierls, lines 5–16 of Algorithm 1 perform  $\mathcal{O}(\tilde{I})$  operations. This number of operations is multiplied by each operation's cost (from Lemma 4.4), yielding the complexity  $\mathcal{O}(\tilde{I}[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}])$ .

The above arguments yield the complexity of Algorithm 1 as  $\mathcal{O}(|A(:, k)| + \tilde{I} +$

$|\mathcal{X}| \log |\mathcal{X}| + |\mathcal{X}| + \tilde{I}[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}]$ ). Since  $|A(:, k)| \leq |\mathcal{X}| \leq \tilde{I}$  (lines 5–16 perform at least one operation on each entry in  $\mathcal{X}$ ), the overall complexity is reduced to  $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}| + \tilde{I}[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}])$ . Finally, we show that  $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|) \subseteq \mathcal{O}(\tilde{I}[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}])$  via an expansion of the cost of the integer arithmetic operations (from inequality (10)):

$$\mathcal{O}(\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}) = \mathcal{O}(n \log \sigma \sqrt{\gamma} \log(n \log \sigma \sqrt{\gamma}) \log \log(n \log \sigma \sqrt{\gamma})).$$

Since  $|\mathcal{X}| \log |\mathcal{X}| \leq n \log n$ , the complexity of the overall sparse REF triangular solve becomes  $\mathcal{O}(\tilde{I}[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}])$ .  $\square$

**4.2.2. SLIP LU factorization complexity.** This subsection presents the complexity of SLIP LU. Theorem 4.8 formally presents the complexity of the SLIP LU factorization and shows that its computational complexity is indeed proportional to its arithmetic work.

**THEOREM 4.8.** *The computational complexity of the SLIP LU factorization is  $\mathcal{O}(I[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}])$ .*

*Proof.* The complexity given in Theorem 4.8 follows from the previous subsection. In Theorem 4.7, we proved that the sparse REF triangular solve's complexity is proportional to the number of integer arithmetic operations performed. Thus, since the overall factorization consists only of repeated triangular solves, the complexity is the total number of operations performed,  $I$ , multiplied by their cost. In addition, at the beginning of the factorization, the  $\mathcal{X}$ ,  $\mathbf{h}$ , and  $\mathbf{x}$  vectors are initialized taking  $\mathcal{O}(n)$  time. This analysis yields a complexity of  $\mathcal{O}(n + I[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}])$ . Finally, we note that the SLIP LU factorization must perform at least one operation for each entry along the diagonal of the matrix  $A$  (lines 6–8 in Algorithm 1); therefore,  $\mathcal{O}(n) \subseteq \mathcal{O}(I)$  (with equality if and only if  $A$  is diagonal), which yields the complexity of the SLIP LU factorization as  $\mathcal{O}(I[\beta_{\max} \log \beta_{\max} \log \log \beta_{\max}])$ .  $\square$

**4.2.3. Relationship between the complexity of SLIP LU and left-looking LU.** Comparing the floating point operations in Gilbert and Peierls' triangular solve (page 865 of [25]) to the integer arithmetic operations in Algorithm 1, we see that the sparse REF triangular solve has an additional operation in the outer for loop (the History update). Therefore, in general,  $\mathcal{O}(f) \neq \mathcal{O}(I)$ . This fact can be most easily seen by considering a diagonal matrix. In this case, LLU sets  $U = A$  and  $L = I$ ; therefore,  $f = 0$ . Conversely, SLIP LU must perform History updates along the diagonal, meaning that  $I = 2(n - 1)$ ; thus obviously  $\mathcal{O}(f) \neq \mathcal{O}(I)$ .

However, if for each nonzero  $u_{j,k}$ , there is at least one nonzero  $l_{i,j}$  such that  $i > j$ , then  $\mathcal{O}(f) = \mathcal{O}(I)$ . We can prove this statement via the following observations. First, assuming identical row and column permutations, the nonzero patterns of  $L$  and  $U$  are identical for both SLIP LU and LLU. This follows from the fact that Escobedo, Moreno-Centeno, and Lourenco proved that the Doolittle  $L$  and  $U$  matrices can be obtained by dividing the REF LU  $L$  and  $U$  matrices by the principal minors of  $A$  (i.e., REF LU's pivots) [20]. Second, subsection 3.3.1 showed that the nonzero pattern of each column,  $\mathcal{X}$ , is identical for both factorizations. Third, the cost of the History update (lines 6–8) can be amortized as follows. For nonzeros in  $L$ , since the nonzero patterns of both factorizations'  $L$  matrices are the same, History updates in SLIP LU are identical to pivot operations in LLU. For nonzeros in  $U$ , if the above condition holds, the cost of each History update can be amortized to the cost of the inner IPGE update.

**5. Computational results.** This section illustrates the computational superiority of SLIP LU over a state-of-the-art rational arithmetic left-looking LU factorization for a test set of real world LP basis matrices. Subsection 5.1 details the specifications of the computational study. Subsection 5.2 describes the heuristics utilized for column and row ordering. Finally, subsection 5.3 presents a detailed computational comparison of the two factorization algorithms.

### 5.1. Specifications.

**5.1.1. Rational LU factorization.** The software package `CSparse` [9] is a modern implementation of the LLU factorization in floating point arithmetic. Although written for illustrating sparse matrix algorithms as a textbook software package [11], `CSparse` is highly robust, with no bugs encountered since its publication in 2006. It is also widely used in industry; for example, it forms the basis of `dmperm` in MATLAB and is part of the built-in libraries in several smartphone operating systems. To make `CSparse` exact, we removed all numerical precautions and modified the calculations within its LU factorization subroutines from fixed precision floating point to full-precision rational arithmetic. This conversion was done using the GNU Multiple Precision (GMP) Arithmetic Library [28], a C/C++ numeric library that provides efficient algorithms for exact integer and rational arithmetic operations. Henceforth, this exact version of `CSparse` will be referred to as QLU.

We note that QLU serves as strong competition to our algorithm for the following reasons. First, this approach of modifying the internal operations to exact arithmetic mirrors the approach used in the exact LP solvers `QSopt` [2] and `Soplex` [26]—the state-of-the-art solvers for exact LP. Second, we ran a set of preliminary computational results comparing QLU to an exact version of the C++ solver Eigen [32]. The Eigen solver was made exact via the GMP C++ interface as a template argument into Eigen. These computational results showed that QLU dramatically outperformed this rational version of Eigen, thereby illustrating that QLU serves as an exact, commercial quality implementation of LLU. The results of these tests are available upon request.

**5.1.2. Computing environment.** The experiments were performed using a computing node which has 22 GB of RAM shared by two 2.8 GHz quad core Intel Xeon 5560 processors. All full-precision integer and rational arithmetic operations were performed using the GMP library [28].

**5.1.3. Chosen instances.** We tested the factorizations on the `BasisLIB_INT` repository, a set of 276 final LP bases and corresponding right-hand side vectors obtained as output from the `QSopt_ex` LP solver [8]. The matrices within this repository range in dimension from 100 to 50,000 rows and columns, and have been scaled so that all numeric entries are integral. `BasisLIB_INT` was chosen for two reasons: (1) it is a collection of real world sparse matrices arising from LP applications, and (2) the majority of the matrices are very ill-conditioned (i.e., 51% of the matrices have an estimated condition number exceeding  $10^8$ ). Thus, this collection represents a set of matrices that are both practical and prone to the numerical issues that can cripple floating point solvers. The `BasisLIB_INT` repository is hosted by the Sparse Integer Matrices Collection and is freely available at <http://hpac.imag.fr/Matrices/BasisLIB/>.

**5.2. Ordering schemes.** As briefly mentioned in the introduction, a key step of sparse LU factorization algorithms is to find row and column permutation matrices ( $P$  and  $Q$ , respectively) so that the factorization of the permuted matrix  $PAQ$  requires less work than the direct factorization of  $A$ . Yannakakis showed that finding

the optimal  $P$  and  $Q$  to minimize fill-in is NP-Complete [48]. Moreover, the permutation which minimizes the total number of arithmetic operations is also NP-Hard [38]. Therefore, in practice, various heuristics are used to obtain these permutation matrices. Moreover, no specialized row and column orderings have been developed for exact methods. This subsection discusses the chosen heuristics to obtain these row and column orderings.

**5.2.1. Column preordering.** The column ordering in LLU is typically chosen and fixed prior to the factorization. In floating point arithmetic, many promising column orderings are based on the general purpose minimum degree algorithm, in which pivot columns are selected by minimizing the external row degree. However, computing these degrees exactly is costly in practice; therefore, the three most widely used approximations, approximate minimum degree (AMD) [1], column minimum degree (COLMMD) [24], and column approximate minimum degree (COLAMD) [13], minimize a relaxation of this external row degree. In a computational study of AMD, COLMMD, and COLAMD on a set of square unsymmetric matrices, Davis et al. [13] showed that COLAMD typically produces the column ordering which results in the least fill-in.

Although these column ordering algorithms were designed with floating point arithmetic in mind, it is reasonable to expect less fill-in will result in faster factorization times for both SLIP LU and QLU, as minimizing fill-in is related to reducing the number of arithmetic operations [38]. Therefore, since the matrices within the BasisLIB\_INT repository are square and unsymmetric, we utilize the COLAMD column preorder in all ensuing analyses. The functions to perform COLAMD were obtained via the COLAMD routines within the SuiteSparse C/C++ callable library [12, 10].

**5.2.2. Pivoting schemes.** At iteration  $k$ , LLU does not have access to the submatrix of  $A$  (i.e., rows and columns  $k + 1 : n$  of  $A$ ); therefore, it cannot select pivots based on a sparsity criterion. Instead, the traditional approach for unsymmetric matrices is to select the  $k$ th pivot via a partial pivoting scheme in which the entry of largest magnitude in the  $k$ th column is chosen [25]. Though, in floating point arithmetic, this is done to preserve numerical stability, a variant of this strategy is also used in the rational arithmetic LU factorization routines within the exact LP solvers QSOPT\_ex [3, 2] and Soplex-exact [26, 46]. Therefore, since selecting the largest pivot is standard in right-looking rational arithmetic LU factorizations, we use this strategy in QLU.

Conversely, it remains an open question as to which pivoting criterion works best for IPGE based algorithms. Indeed, Lee and Saunders state that “no clear choice is evident” with respect to pivoting strategies [37]. Based on the complexity of SLIP LU given in subsection 4.2.2, two factors influence the worst-case run time: (1) number of operations to be performed, and (2) size of each number (e.g.,  $\beta_{max}$ ). Therefore, since the COLAMD column ordering aims to reduce the number of operations, it is reasonable to select a pivoting strategy which reduces the size of entries in the factorization. To do this, we utilize a partial pivoting scheme in which the *smallest* entry in the  $k$ th column is selected as the  $k$ th pivot. This choice is motivated as follows. Recall that each iteration consists of a sequence of IPGE and History updates. These updates include a multiplication by the current pivot and a division by a previous pivot. Also, since the  $k$ th pivot is a  $k \times k$  subdeterminant of the input matrix  $A$ , its size (based on Hadamard’s bound [31]) is bounded above by

$$\rho^k \leq (k\sigma^2)^{\frac{k}{2}}.$$

Therefore, since the worst-case size of the  $k$ th pivot dominates the worst-case size of all previous pivots, selecting the smallest pivot at each iteration should help reduce the growth of entries in the Schur complement.

**5.3. QLU versus SLIP LU.** This subsection compares the performances of QLU and SLIP LU across the aforementioned 276 matrices from the `BasisLIB_INT` repository. In the tables in this section, the following abbreviations are used: **AVG**: arithmetic mean in seconds, **GM**: geometric mean, **Best**: number of problems in which the referenced algorithm was faster, **nnz**: number of nonzeros in the matrix, and **nnz/col**: average number of nonzeros per column in the matrix.

This subsection is organized as follows. Subsection 5.3.1 gives a comparison of the two algorithms across all instances for both factorization and forward/backward (FB) substitution. Subsection 5.3.2 relates each algorithm's performance to the fill-in of the  $L$  and  $U$  factors. Subsections 5.3.3 and 5.3.4 highlight specific instances, particularly those in which one algorithm dramatically outperforms the other and those whose factorization run time exceeded one hour for each method. For completeness, Appendix E presents detailed results for the entire repository.

**5.3.1. SLIP LU versus QLU: All instances.** Across the 276 benchmark instances, SLIP LU outperformed QLU in both factorization and FB substitution time. In particular Tables 1 and 2 compare the two algorithms' run times in seconds. Table 1 shows that SLIP LU factorizes 59% of the matrices faster than QLU. Also, Table 1 shows that, for factorization time, SLIP LU has an average and geometric mean 6.1 and 1.5 times smaller than those of QLU, respectively. Likewise, for FB substitution, Table 2 shows that SLIP LU is faster than QLU for 75% of the matrices. Additionally, for FB substitution, SLIP LU has an average and geometric mean 3.4 and 4.0 times smaller than those of QLU, respectively.

TABLE 1  
*Factorization time: In all metrics, SLIP LU outperforms QLU.*

	AVG	GM	Best
SLIP LU	145.64	0.14	164
QLU	888.47	0.21	112

TABLE 2  
*FB substitution time: In all metrics, SLIP LU outperforms QLU.*

	AVG	GM	Best
SLIP LU	5.01	0.02	208
QLU	17.19	0.08	68

Graphically, the superiority of SLIP LU is evident by the performance profiles shown in Figure 1. For the performance profiles, all factorization and FB substitution times are shifted by one second (therefore, dramatic yet intranscendental differences such as 1s versus 0.1s will not produce misleading results). Figure 1 shows that SLIP LU dominates QLU for factorization time and FB substitution time. Overall, these performance profiles illustrate the clear superiority of SLIP LU over rational LU. Since factorization time dominates the FB substitution time, the ensuing in-depth analyses focus solely on factorization time.

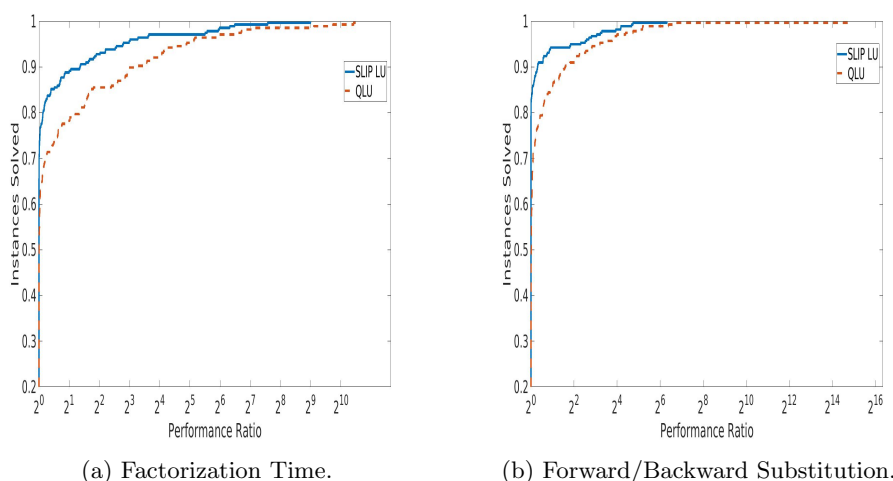
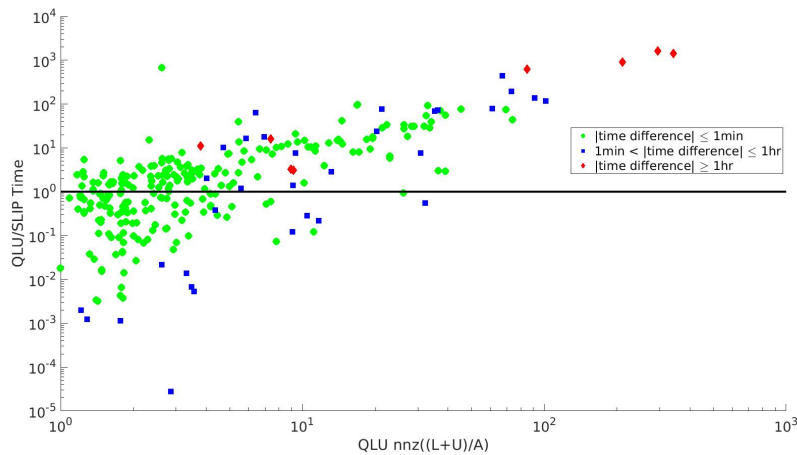
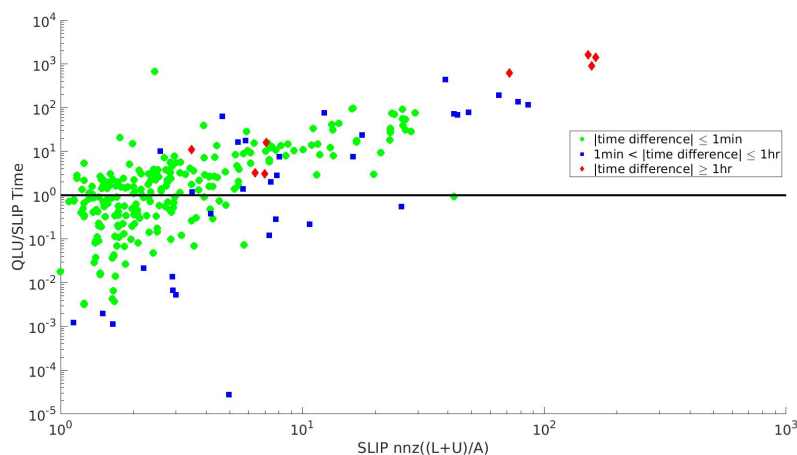


FIG. 1. *SLIP LU dominates QLU in both factorization and FB substitution times.*

**5.3.2. Run times compared to fill-in.** This section analyzes the relationship of the factorization time ratios to the amount of fill-in that occurs during the factorization. The factorization time ratio is computed as the run time of QLU divided by the run time of SLIP LU (thus if this ratio exceeds 1, SLIP LU is superior, and if it is less than 1, QLU is superior). The amount of fill-in is computed as the number of nonzeros in  $L + U$  divided by the number of nonzeros in  $A$ . Figures 2 and 3 plot the factorization time ratio versus the fill-in of QLU and SLIP LU, respectively. Each dot on each plot represents a single matrix from the `BasisLIB_INT` repository. The matrices are color coded as follows: (1) green (circle) if the time difference is less than one minute, (2) blue (square) if the time difference is between one hour and one minute, or (3) red (diamond) if the time difference exceeds one hour, where the time difference is computed as the absolute difference of run time between SLIP LU and QLU. Last, both plots have a line running through the ratio of 1, meaning that any matrix lying above this line is a win for SLIP LU, and any matrix below this line is a win for QLU.

From these plots, we see that SLIP LU regularly outperforms QLU as fill-in increases. Additionally, it is interesting that many of the best ratios for SLIP LU are colored red (meaning that SLIP LU was at least one hour faster than QLU). Similarly, SLIP LU dramatically outperformed QLU for those matrices which suffered the most fill-in. Last, there is no matrix in which QLU outperforms SLIP LU by an hour or more in run time. Overall, these plots show that, in general, QLU has the best performance in those matrices which have the least fill-in. Conversely, SLIP LU has the best performance in those matrices which have higher fill-in (i.e., those requiring more operations). In order to explain these results in more detail, the next subsection analyzes the matrices in which one method was at least 100 times faster than the other (i.e., those matrices lying below  $10^{-2}$  or above  $10^2$ ).

**5.3.3. Instances with extreme run time disparity.** This section analyzes the matrices in which either SLIP LU or QLU was dramatically faster (by a factor of 100). Table 3 shows the nine matrices in which SLIP LU was at least 100 times

FIG. 2. *SLIP LU dominates QLU with high fill-in: QLU fill. (Figure in color online.)*FIG. 3. *SLIP LU dominates QLU with high fill-in: SLIP LU fill. (Figure in color online.)*

faster than QLU. In this table, we see that for four of the matrices, QLU requires more than an hour of run time while SLIP LU computes the factorization in less than a minute. Similarly, Table 4 shows the ten matrices in which QLU is at least 100 times faster than SLIP LU. QLU factors all ten of these matrices in less than one second while SLIP LU factors 7 in less than two minutes and the other three in 3, 5, and 9 minutes, respectively. In the worst-case for QLU, SLIP LU is 1616 times faster, factoring the matrix `nug20` in 22.78 seconds while QLU requires more than ten hours. Conversely, in the worst-case for SLIP LU, QLU is 5150 times faster, factoring the matrix `maros-r7` in 0.01 seconds while SLIP LU requires approximately 8.5 minutes.

These specific results show that SLIP LU performs better on matrices in which QLU requires hours of run time, while QLU performs best on those matrices in which both algorithms can factor in reasonable time. Moreover, the worst-case behavior of SLIP LU is two to three orders of magnitude lower than the worst-case behavior of

QLU. Given these results, we again see that the SLIP LU factorization is superior in the sense that, at worst-case, the SLIP LU factorization requires ten minutes of run time instead of ten hours for the rational factorization. More strikingly, SLIP LU can factor the entire repository *twice* in the time it takes QLU to factor only three matrices: nug20, nug30, and nug15.

TABLE 3  
*Best factorization time ratios for SLIP LU.*

Matrix name	n	nnz	nnz/col	SLIP LU	QLU	QLU/SLIP
nug20	7733	31455	4.1	22.78	36832.23	1616.87
nug30	14681	45627	3.1	23.97	34264.57	1429.48
nug15	5486	24736	4.5	14.75	13398.40	908.37
maros	289	1143	4.0	0.02	12.76	638.00
NSR8K	5387	46157	8.6	6.92	4352.01	628.90
siena1	1265	11573	9.2	0.46	205.17	446.02
rail4284	2463	11802	4.8	0.87	167.66	192.71
nug12	2736	12037	4.4	4.03	548.35	136.07
qap12	2740	12014	4.4	7.88	922.55	117.07

TABLE 4  
*Best factorization time ratios for QLU.*

Matrix name	n	nnz	nnz/col	SLIP LU	QLU	SLIP/QLU
maros-r7	1350	31923	23.6	515.02	0.01	5150.20
scfxm1-2r-256	11812	44985	3.8	66.82	0.08	835.25
jendrec1	1779	34196	19.2	247.57	0.30	825.23
cont111	58936	179556	3.0	107.60	0.21	512.38
watson_1.pre	4642	12991	2.8	7.12	0.02	356.00
watson_1	5729	14544	2.5	8.98	0.03	299.33
scfxm1-2r-128	5671	21943	3.9	11.77	0.04	294.25
scfxm1-2b-64	5966	22682	3.8	12.08	0.05	241.60
mod2.pre	4422	12914	2.9	137.28	0.73	188.05
scfxm1-2r-96	4504	17205	3.8	6.23	0.04	155.75
mod2	4435	12985	2.9	103.91	0.71	146.35

**5.3.4. Instances exceeding one hour.** This section identifies the most time consuming instances for both algorithms (i.e., those exceeding one hour of run time). Table 5 shows the five instances in which SLIP LU exceeded one hour of run time. It is notable that, though SLIP LU required at least one hour for each of these matrices, it still outperforms QLU for three of the five instances. Table 6 shows the eight instances which QLU exceeded one hour of run time. Notably, SLIP LU outperforms QLU in all eight of these matrices, and QLU requires more than eight hours for five of the matrices. Altogether, SLIP LU factorizes *all* five of its worst instances faster than QLU factorizes a single of its five worst instances. Moreover, SLIP LU can factor and solve *all* linear systems within the repository before QLU can factor either of the matrices self or gen1.

**6. Conclusion.** This paper develops a novel factorization algorithm, termed SLIP LU, which exactly solve sparse SLEs in time proportional to its arithmetic work subject to the same caveat of [25], namely that every sparse LU factorization must use heuristics to solve the NP-Hard problem of finding a fill-reducing ordering to minimize the work in the ensuing factorization. This paper provides an efficient sparse adaptation of the REF LU factorization framework and is also the first to

TABLE 5  
SLIP LU: 5 instances exceeding one hour.

Matrix name	n	nnz	nnz/col	SLIP LU	QLU	QLU/SLIP
pilot87.pre	1540	30916	20.1	9705.24	30239.33	3.12
pilot87	1625	31396	19.3	6598.49	21726.50	3.29
gen4	375	8919	23.8	4488.65	977.40	0.22
self	924	157411	170.4	4178.76	45889.35	10.98
gen4.pre	367	9322	25.4	4041.09	1154.83	0.29

TABLE 6  
QLU: 8 instances exceeding one hour.

Matrix name	n	nnz	nnz/col	SLIP LU	QLU	QLU/SLIP
self	924	157411	170.4	4178.76	45889.35	10.98
gen1	329	11016	33.5	2662.68	42778.37	16.07
nug20	7733	31455	4.1	22.78	36832.23	1616.87
nug30	14681	45627	3.1	23.97	34264.57	1429.48
pilot87.pre	1540	30916	20.1	9705.24	30239.33	3.12
pilot87	1625	31396	19.3	6598.59	21726.50	3.29
nug15	5486	24736	4.5	14.50	13398.4	924.03
NSR8K	5387	46157	8.6	6.92	4352.01	628.90

offer a fully sparse implementation of IPGE. Beyond the factorization, this work also updates the IPGE maximum word length for both sparse and positive definite matrices. Computationally, this work compares the SLIP LU factorization with a modern implementation of rational-arithmetic left-looking LU. From this study, we see that the SLIP LU factorization dramatically outperforms the rational factorization in the following ways. First, on a test set of real world sparse linear systems, SLIP LU is faster than rational LU in both average and geometric mean of run time for both factorization and forward/backward substitution. A similar analysis via performance profiles shows that the SLIP LU factorization conclusively dominates the rational factorization. Additionally, we show that as fill-in (i.e., work required) increases, the SLIP LU factorization outperforms rational LU. Finally, we show that SLIP LU dramatically reduces the run times of instances in which the rational factorization requires many hours of run time. Altogether these results present compelling evidence that SLIP LU is preferable to rational-arithmetic left-looking LU factorization for solving real world, unsymmetric sparse linear systems. The code associated with the SLIP LU factorization is hosted at [https://github.com/clouren/SLIP\\_LU](https://github.com/clouren/SLIP_LU).

**Appendix A. Proof of Lemma 3.2.** This section proves the correctness of Lemma 3.2 which states the following.

LEMMA 3.2. *For any  $k \in \{1, \dots, n\}$ , assume the partial matrix  $L^{(k-1)}D^{(k-1)}$  is correct. Then, solving the lower triangular system given by (12) yields the following elements of the REF LU factorization:*

$$x_j = \begin{cases} u_{j,k} & \text{for } j = 1, \dots, k-1, \\ \rho^k = u_{k,k} = l_{k,k} & \text{for } j = k, \\ l_{j,k} & \text{for } j = k+1, \dots, n. \end{cases}$$

We also present

$$(12) \quad L^{k-1} D^{k-1} \mathbf{x} = \left[ \begin{array}{cccc|c} \frac{1}{\rho^0} & 0 & 0 & 0 & \mathbf{0} \\ \frac{l_{2,1}}{\rho^1} & \frac{1}{\rho^1} & 0 & 0 & \\ \vdots & & \ddots & & \\ \frac{l_{k-1,1}}{\rho^1} & \dots & \dots & \frac{1}{\rho^{k-2}} & \\ \hline \frac{l_{k,1}}{\rho^1} & \dots & \dots & \frac{l_{k,k-1}}{\rho^{k-2}\rho^{k-1}} & \\ \vdots & & & \vdots & \\ \frac{l_{n,1}}{\rho^1} & \dots & \dots & \frac{l_{n,k-1}}{\rho^{k-2}\rho^{k-1}} & \frac{1}{\rho^{k-1}} \mathbf{I} \end{array} \right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{1,k}^0 \\ a_{2,k}^0 \\ \vdots \\ a_{n,k}^0 \end{bmatrix},$$

where  $L^k$  is defined as in subsection 2.3 and  $D^k$  is the first completed  $k$  columns of  $D$  augmented by the last  $n - k$  columns of  $(1/\rho^k)I$ .

*Proof.* The proof proceeds in two steps. First, we derive an expression for the  $k$ th column of  $L$  and  $U$ , then we show that this derived expression solves (12).

*Step 1.* An expression for the  $k$ th column of  $L$  and  $U$  can be obtained from the full REF LU factorization. Particularly, in (5), notice that multiplying the matrix  $LD$  by the vector  $U(:, k)$  yields the  $k$ th column of  $A$  as follows:

$$(LD)U(:, k) = \begin{bmatrix} \frac{1}{\rho^0} & 0 & 0 & 0 & \dots & 0 \\ \frac{l_{2,1}}{\rho^1} & \frac{1}{\rho^1} & 0 & 0 & \dots & 0 \\ \frac{l_{3,1}}{\rho^1} & \frac{l_{3,2}}{\rho^1\rho^2} & \frac{1}{\rho^2} & 0 & \dots & 0 \\ \vdots & \vdots & & \ddots & \ddots & \vdots \\ \vdots & & & & \ddots & \vdots \\ \frac{l_{n,1}}{\rho^1} & \frac{l_{n,2}}{\rho^1\rho^2} & \frac{l_{n,3}}{\rho^2\rho^3} & \dots & \dots & \frac{1}{\rho^{n-1}} \end{bmatrix} \begin{bmatrix} u_{1,k} \\ \vdots \\ u_{k-1,k} \\ \rho^k \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} a_{1,k}^0 \\ a_{2,k}^0 \\ \vdots \\ a_{n,k}^0 \end{bmatrix}.$$

By multiplying the matrix  $LD$  by  $U(:, k)$ , we obtain expressions for  $L(:, k)$  and  $U(:, k)$  as follows:

$$(13) \quad \begin{aligned} u_{1,k} &= a_{1,k}^0, \\ u_{j,k} &= \rho^{j-1} \left( a_{j,k}^0 - \sum_{i=1}^{j-1} \frac{l_{j,i} u_{i,k}}{\rho^{i-1} \rho^i} \right) \quad \text{for } 2 \leq j < k, \end{aligned}$$

$$(14) \quad \rho^k = \rho^{k-1} \left( a_{k,k}^0 - \sum_{i=1}^{k-1} \frac{l_{k,i} u_{i,k}}{\rho^{i-1} \rho^i} \right) \quad \text{for } j = k,$$

$$(15) \quad l_{j,k} = \rho^{k-1} \left( a_{j,k}^0 - \sum_{i=1}^{k-1} \frac{l_{j,i} u_{i,k}}{\rho^{i-1} \rho^i} \right) \quad \text{for } j > k.$$

*Step 2.* Now we show that the above expressions are the solution of (12). To do this, we solve (12) (via forward substitution) and obtain

$$(16) \quad x_j = \rho^{\min(j,k)-1} \left( a_{j,k}^0 - \sum_{i=1}^{\min(j,k)-1} \frac{l_{j,i} x_i}{\rho^{i-1} \rho^i} \right).$$

Now, we show that  $x_j = u_{j,k}$  for  $j = 1, \dots, k-1$  via induction on  $j$ .

Base case:  $j = 1$ . Equation (16) reduces to  $x_1 = \rho^0 a_{1,k}^0$ . Since  $\rho^0 = 1$  and  $u_{1,k} = a_{1,k}^0$  (see (6)), then  $x_1 = u_{1,k}$ .

Induction hypothesis. For any  $j < k$ , assume  $x_i = u_{i,k}$  for  $i \leq j-1$ . Then, for  $x_j$ , applying the induction hypothesis to (16) yields

$$(17) \quad x_j = \rho^{j-1} \left( a_{j,k}^0 - \sum_{i=1}^{j-1} \frac{l_{j,i} x_i}{\rho^{i-1} \rho^i} \right) = \rho^{j-1} \left( a_{j,k}^0 - \sum_{i=1}^{j-1} \frac{l_{j,i} u_{i,k}}{\rho^{i-1} \rho^i} \right).$$

Equation (17) is identical to (13); therefore, indeed  $x_j = u_{j,k}$  for  $j = 1, \dots, k-1$ .

Next, we show that  $x_j = l_{j,k}$  for  $j = k+1, \dots, n$ . For an arbitrary  $j > k$ , (16) becomes

$$x_j = \rho^{k-1} \left( a_{j,k}^0 - \sum_{i=1}^{k-1} \frac{l_{j,i} x_i}{\rho^{i-1} \rho^i} \right).$$

We showed  $x_i = u_{i,k}$  for  $i < k$ ; therefore, we have

$$(18) \quad x_j = \rho^{k-1} \left( a_{j,k}^0 - \sum_{i=1}^{k-1} \frac{l_{j,i} u_{i,k}}{\rho^{i-1} \rho^i} \right).$$

Equation (18) is equivalent to (15); thus,  $x_j = l_{j,k}$  for  $j > k$ .

The proof is complete by noting that substituting  $j = k$  into either (17) or (18) yields (14); thus  $x_k = \rho^k$ .  $\square$

**Appendix B. Proof of Theorem 3.1.** This section proves the correctness of Theorem 3.1 which gives the structure of left-looking REF LU as follows.

**THEOREM 3.1** *The REF LU factorization, given by (5) and (6), can be obtained in a left-looking fashion by solving partial lower triangular linear systems of the form  $L^{k-1} D^{k-1} \mathbf{x} = A(:, k)$  for  $k = 1, \dots, n$ :*

$$(19) \quad L^{k-1} D^{k-1} \mathbf{x} = \left[ \begin{array}{cccc|c} \frac{1}{\rho^0} & 0 & 0 & 0 & \mathbf{0} \\ \frac{l_{2,1}}{\rho^1} & \frac{1}{\rho^1} & 0 & 0 & \\ \vdots & & \ddots & & \\ \frac{l_{k-1,1}}{\rho^1} & \dots & \dots & \frac{1}{\rho^{k-2}} & \\ \frac{l_{k,1}}{\rho^1} & \dots & \dots & \frac{l_{k,k-1}}{\rho^{k-2} \rho^{k-1}} & \\ \vdots & & & \vdots & \\ \frac{l_{n,1}}{\rho^1} & \dots & \dots & \frac{l_{n,k-1}}{\rho^{k-2} \rho^{k-1}} & \frac{1}{\rho^{k-1}} \mathbf{I} \end{array} \right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{1,k}^0 \\ a_{2,k}^0 \\ \vdots \\ a_{n,k}^0 \end{bmatrix},$$

where  $L^k$  is defined as in subsection 2.3 and  $D^k$  is the first completed  $k$  columns of  $D$  augmented by the last  $n-k$  columns of  $(1/\rho^k)I$ .

*Proof.* This proof uses induction on  $k$  to show that (19) correctly obtains each column of  $L$  and  $U$ .

Base case:  $k = 1$ . Equation (19) becomes  $\mathbf{I}_{n,n} \mathbf{x} = A(:, 1)$ , meaning  $\mathbf{x} = A(:, 1)$ . Since the first column of REF LU's  $L$  and  $U$  matrices is the entries  $a_{j,1}^0$  for  $j = 1, \dots, n$ , (19) correctly obtains the first column of  $L$  and  $U$ .

Induction hypothesis: For all  $j < k$ , assume that the solution of the system  $L^{k-1}D^{k-1}\mathbf{x} = A(:, j)$  gives the  $j$ th column of  $L$  and  $U$ . Then, for column  $k$ , the induction hypothesis tells us the matrix  $L^{k-1}D^{k-1}$  is correct. Therefore, Lemma 3.2 shows that, since  $L^{k-1}D^{k-1}$  is correct, (19) correctly solves for the  $k$ th column of REF LU. Since this is true for an arbitrary  $k$ , it is true for all  $k$ .  $\square$

**Appendix C. Proof of Theorem 4.2.** This section presents the proof of Theorem 4.2 stated below.

**THEOREM 4.2** *In a sparse matrix, the maximum bit length of any IPGE entry is upper-bounded as*

$$(20) \quad \beta_{max} \leq \lceil n \log(\sigma \sqrt{\gamma}) \rceil.$$

*Proof.* Using Hadamard's bound [29, 31], the maximum value of the determinant of a matrix  $A$  is upper-bounded

$$\text{via a row expansion:} \quad |\det(A)| \leq \prod_{i=1}^n \left( \sum_{j=1}^n |a_{i,j}|^2 \right)^{1/2},$$

$$\text{and via a column expansion:} \quad |\det(A)| \leq \prod_{j=1}^n \left( \sum_{i=1}^n |a_{i,j}|^2 \right)^{1/2}.$$

Note that in at least one of the inequalities, the inner summation can have at most  $\gamma$  nonzero terms. Additionally, each  $a_{i,j} \leq \sigma$ ; thus we have

$$|\det(A)| \leq \prod_{i=1}^n \left( \sum_{j=1}^{\gamma} \sigma^2 \right)^{1/2} = \prod_{i=1}^n \gamma^{1/2} \sigma = \gamma^{n/2} \sigma^n.$$

Since  $\beta_{max}$  is bounded above by  $\log |\det(A)|$ , taking the logarithm of the above term yields inequality (20).  $\square$

**Appendix D. Proof of Theorem 4.3.** This section proves the correctness of Theorem 4.3 stated below.

**THEOREM 4.3** *In an SPD matrix, the maximum bit length of any IPGE entry is upper-bounded as*

$$(21) \quad \beta_{max} \leq \lceil n \log \sigma \rceil.$$

*Proof.* The proof is similar to that of Theorem 4.2. For an SPD matrix  $A$ , the following sequence of inequalities provides a valid upper bound on  $|\det(A)|$ :

$$|\det(A)| \leq \prod_{i=1}^n a_{i,i} \leq \prod_{i=1}^n \sigma = \sigma^n,$$

where the first inequality is Hadamard's bound for SPD matrices [31], and the second inequality follows from the fact that each  $a_{i,i} \leq \sigma$ . Therefore, taking the logarithm of the above expression yields inequality (21).  $\square$

**Appendix E. Computational results for all instances.** Table 7 presents exhaustive computational results for all 276 tested instances. The first three columns give the name, dimension, and number of nonzeros of each matrix. Columns 4 and 5 give the factorization time of SLIP LU and QLU, respectively. Finally, columns 6 and 7 give the forward and backward substitution time for each algorithm. All times are in seconds and rounded to three decimal places.

Matrix name	n	nnz	Factorization (s)		FB substitution (s)	
			SLIP LU	QLU	SLIP LU	QLU
10teams	177	885	0.017	0.144	0.001	0.070
25fv47	416	2061	0.195	0.229	0.058	0.098
30_70_4.5_0.5_100	2098	6197	0.008	0.024	0.003	0.010
30_70_4.5_0.95_100	2754	8381	0.015	0.050	0.005	0.017
30_70_4.5_0.95_98	2451	7364	0.010	0.025	0.003	0.015
80bau3b	154	396	0.003	0.001	0.004	0.006
aa01	630	4187	0.065	1.775	0.001	0.061
aa03	562	3420	0.031	0.552	0.001	0.015
air04	630	4187	0.052	1.753	0.001	0.026
air05	323	1789	0.015	0.185	0.001	0.007
air06	562	3420	0.033	0.541	0.001	0.015
arki001	160	893	0.356	0.032	0.208	0.105
bandm	122	609	0.010	0.011	0.003	0.010
bas1lp	502	6651	0.022	0.884	0.002	0.051
baxter	256	697	0.003	0.004	0.001	0.003
baxter.pre	470	1274	0.012	0.002	0.002	0.003
bg512142	560	2140	0.036	0.150	0.013	0.085
biella1	813	5726	0.037	0.396	0.002	0.023
bienst1	102	253	0.000	0.000	0.000	0.001
blp-ar98	148	876	0.001	0.002	0.000	0.002
bnl1	223	824	0.006	0.003	0.004	0.004
bnl2	459	1488	0.009	0.005	0.007	0.005
boeing1	122	415	0.001	0.002	0.001	0.002
brd14051	16360	180847	171.920	1314.110	4.156	35.856
capri	138	507	0.007	0.017	0.007	0.011
car4	122	4384	0.211	3.226	0.012	0.324
ceria3d	130	647	0.001	0.002	0.000	0.001
ch	393	1304	0.082	0.005	0.047	0.007
co5	928	6173	2.242	1.032	0.400	0.464
co9	2287	13481	13.055	6.854	1.507	1.956
complex	327	10738	0.215	2.666	0.002	0.046
cont1_l	1070	4649	5.985	1.747	0.600	0.333
cont11_l	58936	179556	107.592	0.214	98.297	0.245
cont4	2802	11862	44.845	21.905	3.471	3.059
cq5	570	2615	0.197	0.044	0.068	0.098
cq9	1187	5786	1.517	0.525	0.502	0.764
cr42	304	608	0.093	0.002	0.221	0.004
crew1	127	861	0.002	0.016	0.000	0.002
cycle	284	878	0.017	0.005	0.002	0.004
d15112	9197	47335	18.072	2.217	4.033	0.410
d18512	10815	55880	1.920	3.102	0.284	0.341

Matrix name	n	nnz	Factorization (s)		FB substitution (s)	
			SLIP LU	QLU	SLIP LU	QLU
d2q06c	1047	5717	18.513	5.659	2.575	4.327
d6cube	223	1424	0.018	0.231	0.001	0.013
dano3mip	1135	5390	0.900	65.526	0.027	1.819
dano3mip.pre	1091	5239	0.932	64.397	0.026	1.828
danoint	196	790	0.006	0.028	0.002	0.011
dbic1	4795	23403	7.036	6.661	0.633	0.103
dbir1	154	845	0.005	0.002	0.003	0.001
dbir2	157	784	0.001	0.005	0.000	0.002
dbir2.pre	281	1879	0.005	0.002	0.001	0.002
dc1c	808	4698	0.079	5.539	0.003	0.105
dc1l	851	5171	0.130	7.192	0.003	0.148
dcmulti	120	303	0.001	0.000	0.000	0.001
de063155	313	1233	0.552	4.011	0.214	0.868
de063157	282	1102	0.282	0.249	0.109	0.148
de080285	368	1493	1.453	3.988	0.289	0.547
degen2	217	1138	0.003	0.023	0.000	0.037
degen3	744	5431	0.026	0.368	0.001	0.014
delf000	593	1606	0.097	0.004	0.099	0.012
df2177	414	1825	0.036	0.531	0.001	0.011
dfl001	3271	9276	0.209	8.305	0.006	0.163
dfl001.pre	2097	6501	0.154	14.246	0.004	0.176
dg012142	892	3627	0.119	0.110	0.017	0.043
disctom	192	565	0.002	0.023	0.000	0.007
dolom1	806	5681	0.080	2.489	0.002	0.073
ds	647	12193	0.093	0.797	0.003	0.048
dsbmip	220	568	0.002	0.001	0.002	0.002
fast0507	401	1908	0.009	0.130	0.001	0.066
fnl4461	5044	46977	2.481	20.200	0.108	1.225
fome11	6226	17749	0.414	22.744	0.013	0.295
fome12	12652	35969	1.957	55.734	0.052	0.701
fome13	24884	70839	174.570	95.639	3.880	1.686
fome20	1718	3811	0.006	0.020	0.001	0.030
fome21	3291	7240	0.008	0.061	0.002	0.013
ganges	344	1123	0.014	0.003	0.010	0.003
ge	1675	4758	1.135	0.030	0.584	0.077
gen1	329	11016	2662.677	42778.367	45.402	1683.507
gen2	328	8894	80.908	618.517	1.436	19.442
gen4	375	8919	4488.647	977.397	57.632	34.341
gen4.pre	367	9322	4041.093	1154.830	55.038	36.939
gesa2_o	102	214	0.001	0.001	0.001	0.022
gesa3_o	148	365	0.001	0.001	0.001	0.002
gesa3	134	336	0.001	0.001	0.001	0.002
gosh	379	1379	0.034	0.006	0.019	0.015
gran	284	1958	0.185	0.020	0.039	0.038
greenbea	664	2706	0.235	0.047	0.087	0.065
greenbeb	713	3278	0.374	0.138	0.096	0.106
grow15	297	3614	0.338	0.523	0.063	0.200
grow22	434	4711	0.825	1.874	0.169	0.712

Matrix name	n	nnz	Factorization (s)		FB substitution (s)	
			SLIP LU	QLU	SLIP LU	QLU
grow7	138	1744	0.069	0.101	0.016	0.067
iiasa	113	262	0.001	0.000	0.001	0.001
jendrec1	1779	34196	247.566	0.303	200.339	6.875
l30	2492	12653	128.521	48.815	4.458	9.556
l30.pre	1199	6030	12.693	38.214	0.005	0.006
l9	241	1381	0.060	0.229	0.007	0.062
large000	823	2282	0.216	0.006	0.306	0.031
lp22	1796	13076	0.378	37.353	0.017	1.055
lp22.pre	1811	13146	0.434	41.733	0.014	1.062
lpl1	2692	7211	0.050	0.065	0.020	0.031
lpl3	212	461	0.001	0.001	0.000	0.001
manna81	1392	2784	0.002	0.005	0.002	0.004
maros-r7	1350	31923	515.015	0.014	8.713	0.038
maros	289	1143	0.019	12.756	0.005	13.664
mitre	801	2466	0.019	0.007	0.013	0.005
mkc	106	250	0.001	0.001	0.000	0.001
mkc1	106	250	0.000	0.001	0.000	0.003
mod2	4435	12985	103.907	0.706	41.793	1.780
mod2.pre	4422	12914	137.280	0.729	46.345	1.610
model10	1341	6403	25.035	65.132	3.460	14.296
model11	2039	7606	5.301	0.367	0.742	0.153
model2	149	757	0.042	0.056	0.014	0.049
model3	310	1417	0.085	0.106	0.039	0.085
model4	409	1898	0.606	0.857	0.230	0.648
model5	492	2247	0.874	0.155	0.240	0.233
model6	790	3425	2.308	1.565	0.525	1.199
model7	646	2850	0.690	0.483	0.249	0.535
model9	902	4361	0.807	0.119	0.156	0.153
modszk1	263	765	0.022	0.003	0.008	0.004
momentum1	932	2792	0.267	0.167	0.233	0.322
momentum2	2113	6516	8.319	0.617	0.903	0.059
momentum3	3254	15159	321.933	383.977	38.383	115.663
msc98-ip	2897	10006	0.082	0.094	0.015	0.058
mzzv11	1098	3189	0.004	0.010	0.001	0.031
mzzv42z	787	2124	0.003	0.005	0.001	0.034
nemsemm2	789	2440	0.139	0.011	0.132	0.012
nemspmm1	982	5023	2.106	0.209	0.357	0.096
nemspmm2	949	6478	4.228	1.935	0.756	1.562
nemswrld	2205	13323	53.428	14.100	4.968	3.814
neos	2342	5098	0.003	0.008	0.002	0.006
neos.pre	2080	4578	0.003	0.009	0.002	0.007
neos1	309	944	0.002	0.006	0.000	0.002
neos11	365	1116	0.003	0.010	0.000	0.003
neos19	228	487	0.001	0.001	0.000	0.001
neos4	454	944	0.001	0.001	0.000	0.002
neos6	174	1580	0.003	0.034	0.000	0.004
neos7	590	1434	0.006	0.003	0.004	0.003
neos818918	265	678	0.001	0.003	0.000	0.002

Matrix name	n	nnz	Factorization (s)		FB substitution (s)	
			SLIP LU	QLU	SLIP LU	QLU
neos823206	220	547	0.001	0.001	0.000	0.002
nesm	279	895	0.663	0.095	0.308	0.153
newman	334	2156	1.179	4.006	0.192	0.964
newman2	468	7917	10.867	111.840	0.416	6.540
newman3	369	3662	0.357	1.109	0.039	0.178
nl	890	2919	2.495	0.120	1.070	0.219
nsct1	120	595	0.004	0.001	0.002	0.002
nsct2	107	544	0.001	0.003	0.000	0.002
nsct2.pre	156	1140	0.002	0.001	0.000	0.002
NSR8K	5387	46157	6.924	4352.010	0.093	28.201
nug05	107	362	0.001	0.002	0.000	0.028
nug06	267	1007	0.007	0.092	0.000	0.005
nug07	450	1780	0.051	0.417	0.000	0.011
nug08	732	3004	0.134	2.481	0.001	0.036
nug12	2736	12037	4.030	548.349	0.024	3.292
nug15	5486	24736	14.748	13398.400	0.060	37.725
nug20	7733	31455	22.781	36832.233	0.134	94.004
nug30	14681	45627	23.968	34264.567	0.245	119.551
orna1	810	2842	0.467	0.578	0.321	0.718
p010	839	2486	0.004	0.007	0.001	0.128
p05	919	2717	0.007	0.012	0.002	0.054
p19	117	555	0.002	0.002	0.001	0.004
pcb1000	1156	9955	0.034	0.975	0.004	0.055
pcb3000	3058	27446	0.187	8.217	0.015	0.515
pcb3038	3588	46560	4.175	317.786	0.118	3.634
pds-100	8377	17555	0.019	0.086	0.007	0.018
pds-20.pre	370	851	0.001	0.002	0.000	0.002
pds-30	2643	5641	0.005	0.017	0.002	0.008
pds-40	4028	8478	0.007	0.028	0.003	0.012
pds-50	5962	12592	0.014	0.054	0.004	0.015
pds-60	7586	16067	0.018	0.097	0.004	0.020
pds-70	7822	16545	0.019	0.110	0.004	0.020
pds-80	9225	19432	0.021	0.113	0.005	0.020
pds-90	7914	16673	0.017	0.096	0.004	0.019
perold	440	2584	7.169	11.502	1.160	4.516
pf2177	406	1772	0.084	0.784	0.002	0.042
pilot.ja	567	3781	25.151	1587.247	2.609	56.262
pilot	1132	16624	600.735	73.246	11.418	17.893
pilot.we	554	2367	1.334	2.965	0.712	1.839
pilot4	289	2805	7.736	12.561	1.084	6.288
pilot4i	134	1220	0.047	0.023	0.014	0.024
pilot87	1625	31396	6598.487	21726.500	78.389	731.339
pilot87.pre	1540	30916	9705.240	30239.333	113.564	666.582
pilotnov	549	3337	8.639	23.376	1.142	4.266
pla33810	18940	123445	24.817	14.644	1.117	0.815
pla7397	5059	42683	1.804	18.708	0.098	1.038
pla8_sig185	39835	196256	972.214	2762.003	11.061	33.197
pla85900.nov21	40304	230558	412.146	570.981	8.339	16.669

Matrix name	n	nnz	Factorization (s)		FB substitution (s)	
			SLIP LU	QLU	SLIP LU	QLU
padded000b	537	1448	0.080	0.009	0.155	0.171
pp08aCUTS	131	332	0.001	0.001	0.001	0.002
progas	1167	6500	14.206	12.486	3.436	17.893
protfold	574	2562	0.039	1.337	0.002	0.160
qap10	1510	6381	0.800	63.354	0.008	0.513
qap12	2740	12014	7.877	922.545	0.046	3.952
qiu	603	1717	0.015	0.100	0.003	0.047
qiulp	603	1717	0.018	0.104	0.004	0.018
r05	919	2717	0.006	0.013	0.002	0.010
rail4284	2463	11802	0.870	167.664	0.018	1.526
rail507	413	2005	0.015	0.166	0.001	0.007
rail516	268	936	0.005	0.027	0.000	0.003
rail582	384	1387	0.006	0.065	0.001	0.010
rat1	452	2893	1.063	0.978	0.193	0.395
rat5	902	12026	40.581	725.085	1.477	34.206
rat7a	641	10542	18.802	311.807	0.642	17.227
rd-rplusc-21	148	454	0.001	0.001	0.001	0.002
rentacar	327	1080	0.014	0.020	0.002	0.005
rl11849	6769	40885	2.057	8.118	0.209	0.775
rl5915	3853	28829	0.831	13.077	0.067	1.268
rl5934	3773	23917	0.690	10.740	0.045	0.696
roll3000	177	1101	0.003	0.009	0.001	0.003
rosen1	217	2528	0.009	0.029	0.003	0.015
rosen10	989	6916	0.041	0.031	0.020	0.020
rosen2	431	4143	0.036	0.026	0.007	0.020
rosen7	127	649	0.001	0.002	0.000	0.002
rosen8	264	1850	0.006	0.013	0.002	0.008
route	339	1290	0.003	0.019	0.001	0.005
sc205	184	487	0.001	0.001	0.000	0.002
scagr7-2r-864	680	1697	0.005	0.003	0.011	0.004
scfxm1-2b-16	784	2975	0.118	0.013	0.038	0.022
scfxm1-2b-4	233	965	0.009	0.004	0.004	0.005
scfxm1-2b-64	5966	22682	12.083	0.052	5.372	0.062
scfxm1-2c-4	233	965	0.010	0.003	0.005	0.005
scfxm1-2r-128	5671	21943	11.770	0.045	5.165	0.072
scfxm1-2r-16	752	2962	0.135	0.009	0.042	0.013
scfxm1-2r-256	11812	44985	66.827	0.077	28.940	0.118
scfxm1-2r-27	1222	4753	0.326	0.013	0.140	0.018
scfxm1-2r-32	1447	5658	0.453	0.020	0.200	0.030
scfxm1-2r-4	233	965	0.008	0.005	0.003	0.006
scfxm1-2r-64	1870	11122	2.293	0.033	0.971	0.051
scfxm1-2r-8	403	1608	0.028	0.006	0.015	0.008
scfxm1-2r-96	4504	17205	6.229	0.041	2.704	0.064
scfxm2	178	658	0.003	0.002	0.002	0.003
scfxm3	262	1005	0.005	0.002	0.004	0.003
scorpion	131	507	0.001	0.001	0.000	0.001
scrs8-2c-64	168	336	0.000	0.001	0.000	0.001
scrs8-2r-128	192	384	0.001	0.001	0.000	0.002

Matrix name	n	nnz	Factorization (s)		FB substitution (s)	
			SLIP LU	QLU	SLIP LU	QLU
scrs8-2r-256	416	832	0.001	0.000	0.001	0.001
scrs8-2r-32	128	256	0.000	0.003	0.000	0.002
scrs8-2r-512	992	1984	0.003	0.001	0.002	0.001
scrs8-2r-64	256	512	0.001	0.001	0.001	0.001
scrs8	109	280	0.001	0.001	0.001	0.001
scsd8	247	655	0.019	0.002	0.009	0.002
self	924	157411	4178.763	45889.350	21.801	479.497
seymour	537	1881	0.005	0.038	0.001	0.006
sgpf5y6	787	1870	0.002	0.003	0.001	0.005
sgpf5y6.pre	755	1744	0.002	0.004	0.000	0.004
sienal	1265	11573	0.464	205.175	0.007	1.207
slptsk	2315	34430	3289.083	71.565	320.334	63.725
small000	140	383	0.003	0.001	0.004	31.490
south31	112	460	0.012	0.004	0.020	0.041
sp97ar	271	2400	0.005	0.023	0.001	0.025
sp98ar	223	1782	0.004	0.015	0.001	0.005
stair	324	3431	0.310	1.028	0.028	0.166
stat96v1	5013	20325	40.449	970.207	1.437	16.101
stat96v2	12928	48009	29.929	87.770	1.597	16.244
stat96v3	13485	49917	25.730	77.991	1.611	10.363
stat96v4	3139	23752	388.395	790.958	42.583	130.835
stat96v5	812	3795	13.042	1.056	3.844	65.299
stocfor2	224	576	0.005	0.002	0.004	0.747
stocfor3	1782	4562	0.333	0.012	0.247	0.011
stormg2_1000	14075	32597	2.901	0.048	5.628	0.028
stormg2_1000.pre	13926	32547	3.274	0.051	6.252	0.035
stormg2-125	1886	4372	0.035	0.008	0.061	0.017
stormg2-125.pre	1780	4138	0.036	0.007	0.053	0.007
stormg2-27	449	1019	0.002	0.002	0.002	0.004
stormg2-8	117	263	0.001	0.001	0.000	0.002
stp3d	10642	25936	0.118	4.900	0.006	0.080
t0331-4l	520	5034	0.573	17.773	0.007	0.344
t1717	549	3657	0.184	5.785	0.006	0.264
trento1	1070	10010	0.151	11.579	0.002	0.122
ulevimin	697	1879	0.240	0.133	0.122	0.072
UMTS	268	828	0.001	0.023	0.000	0.030
usa13509	3595	19919	0.391	0.848	0.058	0.096
van	7417	21681	0.298	22.207	0.012	0.306
watson_1	5729	14544	8.976	0.029	10.138	0.136
watson_1.pre	4642	12991	7.125	0.024	7.544	0.028
watson_2	1011	2703	0.092	0.008	0.070	0.017
woodw	168	589	0.002	0.004	0.001	0.005
world	4261	12190	131.750	1.796	39.561	1.253

Table 7: Results for all instances.

**Acknowledgment.** The authors want to thank the reviewers for their helpful comments which significantly improved the quality of this paper.

## REFERENCES

- [1] P. R. AMESTOY, T. A. DAVIS, AND I. S. DUFF, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905, <https://doi.org/10.1137/S0895479894278952>.
- [2] D. APPLGATE, W. COOK, S. DASH, AND D. ESPINOZA, *QSOPT ex*, <http://www.dii.uchile.cl/daespino/QSoptExact.doc/main.html>, 2007.
- [3] D. L. APPLGATE, W. COOK, S. DASH, AND D. G. ESPINOZA, *Exact solutions to linear programming problems*, Oper. Res. Lett., 35 (2007), pp. 693–699.
- [4] E. H. BAREISS, *Sylvester's identity and multistep integer-preserving Gaussian elimination*, Math. Comp., 22 (1968), pp. 565–578.
- [5] E. H. BAREISS, *Computational solutions of matrix problems over an integral domain*, J. Inst. Math. Appl., 10 (1972), pp. 68–104.
- [6] B. A. BURTON AND M. OZLEN, *Computing the crosscap number of a knot using integer programming and normal surfaces*, ACM Trans. Math. Software, 39 (2012), 4.
- [7] W. COOK, T. KOCH, D. E. STEFFY, AND K. WOLTER, *An exact rational mixed-integer programming solver*, in International Conference on Integer Programming and Combinatorial Optimization, Lecture Notes in Comput. Sci. 6655, Springer, Heidelberg, 2011, pp. 104–116.
- [8] W. COOK AND D. E. STEFFY, *Solving very sparse rational systems of equations*, ACM Trans. Math. Software, 37 (2011), 39.
- [9] T. A. DAVIS, *CSparse*, in Direct Methods for Sparse Linear Systems, SIAM, Philadelphia, 2006, pp. 145–168, <https://doi.org/10.1137/1.9780898718881.ch9>.
- [10] T. DAVIS, W. HAGER, AND I. DUFF, *SuiteSparse: A Suite of Sparse Matrix Software*, <http://faculty.cse.tamu.edu/davis/suitesparse.html>, 2014.
- [11] T. A. DAVIS, *Direct Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2006, <https://doi.org/10.1137/1.9780898718881>.
- [12] T. A. DAVIS, J. R. GILBERT, S. I. LARIMORE, AND E. G. NG, *Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm*, ACM Trans. Math. Software, 30 (2004), pp. 377–380.
- [13] T. A. DAVIS, J. R. GILBERT, S. I. LARIMORE, AND E. G. NG, *A column approximate minimum degree ordering algorithm*, ACM Trans. Math. Software, 30 (2004), pp. 353–376.
- [14] T. A. DAVIS, S. RAJAMANICKAM, AND W. M. SID-LAKHDAR, *A survey of direct methods for sparse linear systems*, Acta Numer., 25 (2016), pp. 383–566.
- [15] M. DHIFLAOUI, S. FUNKE, C. KWAPPIK, K. MEHLHORN, M. SEEL, E. SCHÖMER, R. SCHULTE, AND D. WEBER, *Certifying and repairing solutions to large LPs how good are LP-solvers?*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 2003, pp. 255–256.
- [16] J. D. DIXON, *Exact solution of linear equations using P-ADIC expansions*, Numer. Math., 40 (1982), pp. 137–141.
- [17] J. EDMONDS, *Systems of distinct representatives and linear algebra*, J. Res. Nat. Bur. Standards Sect. B, 71 (1967), pp. 241–245.
- [18] A. R. ESCOBEDO AND E. MORENO-CENTENO, *Roundoff-error-free algorithms for solving linear systems via Cholesky and Lu factorizations*, INFORMS J. Comput., 27 (2015), pp. 677–689.
- [19] A. R. ESCOBEDO AND E. MORENO-CENTENO, *Roundoff-error-free basis updates of LU factorizations for the efficient validation of optimality certificates*, SIAM J. Matrix Anal. Appl., 38 (2017), pp. 829–853, <https://doi.org/10.1137/16M1089630>.
- [20] A. R. ESCOBEDO, E. MORENO-CENTENO, AND C. LOURENCO, *Solution of dense linear systems via roundoff-error-free factorization algorithms: Theoretical connections and computational comparisons*, ACM Trans. Math. Software, 44 (2018), 40.
- [21] D. G. ESPINOZA, *On Linear Programming, Integer Programming and Cutting Planes*, Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, 2006.
- [22] B. GÄRTNER, *Exact arithmetic at low cost—A case study in linear programming*, Comput. Geom., 13 (1999), pp. 121–139.
- [23] J. R. GILBERT, *Predicting structure in sparse matrix computations*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 62–79, <https://doi.org/10.1137/S0895479887139455>.
- [24] J. R. GILBERT, C. MOLER, AND R. SCHREIBER, *Sparse matrices in MATLAB: Design and*

- implementation, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 333–356, <https://doi.org/10.1137/0613024>.
- [25] J. R. GILBERT AND T. PEIERLS, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 862–874, <https://doi.org/10.1137/0909058>.
  - [26] A. M. GLEIXNER, *Exact and Fast Algorithms for Mixed-Integer Nonlinear Programming*, Ph.D. thesis, Technische Universität, Berlin, Berlin, Germany, 2015.
  - [27] A. M. GLEIXNER, D. E. STEFFY, AND K. WOLTER, *Iterative refinement for linear programming*, INFORMS J. Comput., 28 (2016), pp. 449–464.
  - [28] T. GRANLUND ET AL., *GNU MP 6.0 Multiple Precision Arithmetic Library*, Samurai Media Limited, Wickford, England, 2015.
  - [29] J. HADAMARD, *Résolution d’une question relative aux déterminants*, Bull. Sci. Math, 17 (1893), pp. 240–246.
  - [30] T. C. HALES, *A proof of the Kepler conjecture*, Ann. of Math. (2), 162 (2005), pp. 1065–1185.
  - [31] R. A. HORN AND C. R. JOHNSON, *Matrix Analysis*, Cambridge University Press, Cambridge, UK, 2012.
  - [32] B. JACOB, G. GUENNEBAUD, ET AL., *Eigen: C++ Template Library for Linear Algebra*, [http://eigen.tuxfamily.org/index.php?title=Main\\_Page](http://eigen.tuxfamily.org/index.php?title=Main_Page), 2013.
  - [33] E. KALTOFEN AND B. D. SAUNDERS, *On Wiedemann’s Method of Solving Sparse Linear Systems*, in International Symposium on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes, Lecture Notes in Comput. Sci. 539, Springer, Berlin, 1991, pp. 29–38.
  - [34] E. KLOTZ, *Identification, assessment, and correction of ill-conditioning and numerical instability in linear and integer programs*, in Bridging Data and Decisions, INFORMS, Catonsville, MD, 2014, pp. 54–108.
  - [35] D. E. KNUTH, *The Art of Computer Programming: Sorting and Searching*, Vol. 3, Pearson Education, London, UK, 1998.
  - [36] T. KOCH, *The final NETLIB-LP results*, Oper. Res. Lett., 32 (2004), pp. 138–142.
  - [37] H. R. LEE AND B. D. SAUNDERS, *Fraction free Gaussian elimination for sparse matrices*, J. Symbolic Comput., 19 (1995), pp. 393–402.
  - [38] R. LUCE AND E. G. NG, *On the minimum FLOPs problem in the sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 1–21, <https://doi.org/10.1137/130912438>.
  - [39] R. M. MONTANTE-PARDO AND M. A. MÉNDEZ-CAVAZOS, *Un método numérico para cálculo matricial*, Revista Técnico-Científica de Divulgación, 2 (1977), pp. 1–24.
  - [40] A. NEUMAIER AND O. SCHERBINA, *Safe bounds in linear and mixed-integer linear programming*, Math. Program., 99 (2004), pp. 283–296.
  - [41] F. ORDÓÑEZ AND R. M. FREUND, *Computational experience and the explanatory value of condition measures for linear optimization*, SIAM J. Optim., 14 (2003), pp. 307–333, <https://doi.org/10.1137/S1052623402401804>.
  - [42] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation grosser Zahlen*, Computing, 7 (1971), pp. 281–292.
  - [43] D. E. STEFFY, *Topics in Exact Precision Mathematical Programming*, Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, 2011.
  - [44] Z. WAN, *An algorithm to solve integer linear systems exactly using numerical methods*, J. Symbolic Comput., 41 (2006), pp. 621–632.
  - [45] D. WIEDEMANN, *Solving sparse linear equations over finite fields*, IEEE Trans. Inform. Theory, 32 (1986), pp. 54–62.
  - [46] R. WUNDERLING, *Paralleler und objektorientierter Simplex-Algorithmus*, Ph.D. thesis, Technische Universität Berlin, Berlin, Germany, 1996.
  - [47] R. WUNDERLING, *SoPlex: The Sequential Object-Oriented Simplex Class Library*, 1997, <https://soplex.zib.de/>.
  - [48] M. YANNAKAKIS, *Computing the minimum fill-in is NP-complete*, SIAM J. Algebraic Discrete Methods, 2 (1981), pp. 77–79, <https://doi.org/10.1137/0602010>.