FOUNDATIONAL FACTORIZATION ALGORITHMS FOR THE EFFICIENT ROUNDOFF-ERROR-FREE SOLUTION OF OPTIMIZATION PROBLEMS

A Dissertation

by

ADOLFO RAPHAEL ESCOBEDO

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Erick Moreno-Centeno
Committee Members,	Sergiy Butenko
	Wilbert E. Wilhelm
	Catherine Yan
Head of Department,	César O. Malavé

August 2016

Major Subject: Industrial Engineering

ABSTRACT

LU and Cholesky factorizations play a central role in solving linear and mixed-integer programs. In many documented cases, the roundoff errors accrued during the construction and implementation of these factorizations cause the misclassification of suboptimal solutions as optimal and infeasible problems as feasible and viceversa. Such erroneous outputs bring the reliability of optimization solvers into question and, therefore, it is imperative to eliminate these roundoff errors altogether and to do so efficiently to ensure practicality.

Firstly, this work introduces two roundoff-error-free factorizations (REF) constructed exclusively in integer arithmetic: the REF LU and Cholesky factorizations. Additionally, it develops supplementary integer-preserving substitution algorithms, thereby providing a complete tool set for solving systems of linear equations (SLEs) exactly and efficiently. An inherent property of the REF factorization algorithms is that their entries' bit-length—i.e., the number of bits required for expression—is bounded polynomially. Unlike the exact rational arithmetic methods used in practice, however, the algorithms herein presented do not require any greatest common divisor operations to guarantee this pivotal property.

Secondly, this work derives various useful theoretical results and details computational tests to demonstrate that the REF factorization framework is considerably superior to the rational arithmetic LU factorization approach in computational performance and storage requirements. This is significant because the latter approach is the solution validation tool of choice of state-of-the-art exact linear programming solvers due to its ability to handle both numerically difficult and intricate problems. An additional theoretical contribution and further computational tests also demon-

strate the predominance of the featured framework over Q-matrices, which comprise an alternative integer-preserving approach relying on the basis adjunct matrix.

Thirdly, this work develops special algorithms for updating the REF factorizations. This is necessary because applying the traditional approach to the REF factorizations is inefficient in terms of entry growth and computational effort. In fact, these inefficiencies virtually wipe out all the computational savings commonly expected of factorization updates. Hence, the current work develops REF update algorithms that differ significantly from their traditional counterparts. The featured REF updates are column/row addition, deletion, and replacement.

DEDICATION

To my immediate and extended family who support me with the utmost conviction;

to my mother and Mario who are my model of perseverance and hard work;

to my wife, who is the muse that still laughs at my jokes:

I dedicate this to you all.

ACKNOWLEDGEMENTS

I gratefully acknowledge NSF award CMMI-1252456 for funding this research.

The writing of this dissertation was made possible by the help and inspiration of many great people. First of all, I would like to express my deep gratitude to my advisor, Professor Erick Moreno-Centeno, for tailoring his guidance and mentorship at each major stage of the graduate journey. In particular, it was his meticulous attention that enabled the painstaking transition of my writing style from overelaborate to effective and direct. Most of all, I appreciate his insistence on holding me to high academic standards at every step, his dedicated concern for my success and well-being, and the strong bond that we ultimately developed.

I owe special thanks to the members of my doctoral committee: to Professor Wilbert E. Wilhelm for bestowing me with the know-how and confidence to complete this undertaking and for his kind and generous friendship, to Professor Sergiy Butenko for his open-door policy and for his plethora of professional tips and expertise, and to Professor Catherine Yan for her helpful advice and for her vote of confidence. Additionally, I feel particularly indebted to several professors of my home department for their magnanimity and mentorship at different stages of this long process: Satish T.S. Bukkapatnam, Guy Curry, Alaa Elwany, Thomas Ferris, Mark Lawley, Natarajan Gautam, Kiavash Kianfar, Georgia-Ann Klutke, and César Malavé. You are all the embodiment of the hospitality inherent in the Aggie spirit.

I would like to gratefully recognize the pivotal roles that key faculty outside Texas A&M's walls had toward the completion of my Ph.D. studies. I thank Professor Daphne Der-Fen Liu of Cal State L.A. for kindling my desire to pursue mathematics as an undergraduate and for encouraging and facilitating my success as a graduate

engineering student. I thank Professor Kory Hedman of Arizona State University and Professor Dan Steffy of Oakland University for sharing their respective research expertise and for their enormous support of my professional endeavors. I thank Professor Richard Tapia of Rice University for galvanizing my resolve to "occasionally bark like a big dog" by focusing on performing high-quality research. I also thank Professor William K. Wasson of Cal State L.A. for serving as a model of a civic-minded academician and for his longstanding friendship.

I am highly appreciative of the overwhelming backing and professional training opportunities afforded to me by the Texas A&M LSAMP and AGEP programs. Their support made it possible for me to concentrate on research and to hone the various pivotal skills required to succeed at the next level. In this regard, I thank Dr. Shannon Walton for her leadership in establishing these programs as well as for her candor and good humor. Additionally, I thank Dr. Rhonda Fowler, Isah Veronica D. Juranek, Professor Rosana Moreira, and Dr. Rasheedah Richardson for their expert stewardship of these programs.

I would like to express my effusive gratitude to the many friends I made in the department during the last six years. I owe thanks to Hiram Moya, who was instrumental in my transition to graduate school and whose conversation was always a welcome distraction. I owe thanks to Daniel Jornada and Jose Ramirez for being great study partners and the closest of friends. I also owe special thanks to Michelle Alvarado, Kisuk Sung, and Yaping Wang for their conviviality and camaraderie during my entire time at Texas A&M.

Most importantly, none of this would have been possible without my family's tremendous support. I am unmeasurably grateful to my mother Londy for devoting her life to give me and my siblings the opportunity to succeed in this great country. I am also grateful to her and my stepfather Mario for teaching me a good work ethic by

example. I am grateful to my in-laws Kathy and Bruce for immediately welcoming me to their family and for supporting our stay in Texas through both words and actions. I am grateful to my siblings Nazareth and Stefanie as well as aunts, uncles, cousins, grandparents, and other family members for imposing high expectations on me. I am especially grateful to my wife Amber for pushing me to keep fighting during the most difficult parts of this process and for her constant belief in my abilities. She made countless sacrifices for me to get to this point, and I am not sure if I will ever be able to repay her for all the lost weekends of the last six years. Quite frankly, reaching this milestone would not have been possible without her tireless support and encouragement.

To my professors, colleagues, friends, and family: thank you.

NOMENCLATURE

AVG	Average
ERA	Exact rational arithmetic
gcd	Greatest common divisor
IPGE	Integer-preserving Gaussian elimination
LHS	Left-hand side
LP	Linear programming or linear program, depending on context
MIP	Mixed-integer programming or mixed-integer program, depending on context
REF	Roundoff-error-free
REF-Ch	Roundoff-error-free Cholesky factorization
REF-LU	Roundoff-error-free LU factorization
RHS	Right-hand side
SD	Standard deviation
SLE	System of linear equations
SPD	Symmetric positive definite

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Motivation

Roundoff errors, no matter how seemingly insignificant, can propagate and magnify to a point where they radically affect the output of an algorithm. This snowball effect is especially pronounced for complex and large-scale problems, which are the purview of various interest areas in operations research and engineering. The accumulation of roundoff errors often occurs due to the increased number of iterations and intricate operations required to solve such problems, but it may also result from poor algorithm design and implementation. The most important thing to realize in the latter situation is that the failure to detect and correct the incidence of this accumulation can have very serious consequences. Thus, mathematicians and computer scientists have a crucial responsibility to ensure their computational tools are as immune as possible to the deleterious effects of roundoff errors.

The dangers of accumulating roundoff errors are attested to by historical events (e.g., [19, 55, 85]), perhaps the most cautionary of which is the failure of the Patriot Missile Defense System during Operation Desert Storm. On February 25, 1991, the failure of this system's software prevented an enemy Scud missile from being intercepted, resulting in the death of 28 Americans. According to the official report from the U.S. General Accounting Office [19], the underlying cause was calculation roundoff errors in the system's range-gate algorithm, which is deployed to identify and predict the trajectory of incoming missiles. More specifically, the source of these errors was traced to a subroutine that converts the computer's internal clock counter to "real time". This clock kept track of the tenths of seconds elapsed since the computer was last booted using integers, but real time was kept in seconds using

a floating-point expression. Because the conversion was performed using a 24-bit register and the rational $\frac{1}{10}$ has a non-terminating binary expansion, a very small roundoff error was introduced with each passing second after the computer's last boot time. After 100 hours had elapsed, this resulted in a discrepancy of approximately a third of a second in the calculated time—equating to a shift in the detection range of over 500 meters based on the Scud missile's velocity [6]—which was significant enough to allow the sophisticated defense system to malfunction. Thus, a minor calculation error gradually snowballed into a major disaster.

The preceding analysis of the Patriot Missile Defense System failure brings forth two critical points. First, standard fixed-precision operations lead to roundoff errors that can have serious unforeseen consequences when left unchecked. Second, if the impact of roundoff errors resulting from a simple time conversion subroutine can go largely unnoticed, this raises a concern about the unquestioned deployment of complex roundoff error-prone optimization algorithms to make critical decisions. This concern is grounded on more than mere speculation since roundoff errors have been demonstrated to lead several commercial solvers to misclassify linear programs and mixed-integer linear programs, including very simple ones, as infeasible [51, 64]. Roundoff errors may also cause these solvers to report suboptimal solutions as optimal and infeasible solutions as feasible [20]. It is highly problematic that the aforementioned incongruous solver outputs could potentially serve as the basis for critical decisions encompassing numerous areas of society, including health care, finance, and infrastructure systems. Therefore, the development of viable tools and algorithms that minimize roundoff error, or eliminate it altogether, becomes all the more imperative, given the increasing reliance on mathematical programming software to solve large and complex problems.

The incidence of roundoff error propagation and its potential to derail an algo-

rithm's execution toward highly erratic outcomes is largely unpredictable. Consequently, when solving an optimization problem via roundoff error-prone techniques, there is a possibility that the output may significantly deviate from the correct solution, unbeknownst to the practitioner. We emphasize that this proclamation does not refer simply to minor discrepancies in precision or value relative to the true solution, but rather to the type of deviations that altogether invalidate the solver's optimality, feasibility, or infeasibility certificates, as the previous paragraph explains. Indeed, small errors in solution and objective values can be widely tolerated because approximate solutions that are sufficiently close to the true value are "good enough" in most settings (not to mention that input data of real-world problems are already subject to measurement and human error). On the other hand, an invalid solver outcome—e.g., reporting a suboptimal solution as optimal or an infeasible problem as feasible—could lead to legal, financial, and/or operational problems when implemented. It is also important to point out that there are several applications where even slight deviations from the exact solution are unacceptable. A non-comprehensive list of such applications includes computer-assisted mathematics, combinatorial auctions, health care, and compiler optimization. We refer the reader to Steffy [75] for a detailed description of specific problems within these domains having this stringent requirement.

To highlight the nontrivial levels in solution deviation that may occur when using mathematical programming software, Table 1.1 (adapted from [75]) lists the objective values obtained by applying various solvers—or distinct versions/algorithms of each solver—to the real-world LP instance `sgpf5y6` from the Mittelman LP test set [59]. According to Steffy [75], William Cook and Sanjeeb Dash performed some of the solver runs, while other were retrieved from Mittelman's log files found in [58];

Table 1.1: Objective value deviations for LP instance `sgpf5y6` (adapted from Table 1 in [75])

Solver/Algorithm	**Obj Value**	**Error**
`CPLEX` 7.1 Primal	$-6,398.71$	85.76
`CPLEX` 7.1 Dual	$-6,484.44$	0.03
`CPLEX` 9.0 Primal	$-6,406.78$	77.69
`CPLEX` 9.0 Dual	$-6,484.47$	0.00
`CPLEX` 11.0 Primal	$-6,425.87$	58.60
`CPLEX` 11.0 Dual	$-6,484.46$	0.01
`CPLEX` 12.1 Primal	$-6,425.87$	58.60
`CPLEX` 12.1 Dual	$-6,484.47$	0.00
`CPLEX` 12.4 Primal	$-6,424.23$	60.24
`CPLEX` 12.4 Dual	$-6,441.56$	42.91
`CPLEX` 12.4 Barrier	$-6,460.43$	24.04
`Gurobi` 2.0 Primal	$-6,484.47$	0.00
`Gurobi` 2.0 Dual	$-6,484.47$	0.00
`XPress-15` Primal	$-6,380.45$	104.02
`XPress-15` Dual	$-6,344.30$	140.17
`XPress-20` Primal	$-6,349.93$	134.54
`XPress-20` Dual	$-6,408.02$	76.45
`QSopt` Primal	$-6,419.94$	64.53
`QSopt` Dual	$-6,480.33$	4.14
`CLP-1.02.01`	$-6,480.95$	3.52
`CLP-1.12.0`	$-6,481.26$	3.21
`GLPK-4.37`	$-6,463.66$	20.81
`GLPK-4.44`	$-6,484.47$	0.00
`MOSEK` 6.0	$-6,292.06$	192.41
`SoPlex` 1.2.2	$-6,473.33$	11.14

we performed the `CPLEX`[1] 12.4 runs during the writing of this work using the solver's default settings. All the runs claimed to have obtained the optimal solution, whose exact objective value is given by the rational (–1621116398840608/250000000000), which equals $-6,484.47$ rounding to two decimal places. The corresponding deviations from the exact objective value of each run are stated in the third table column.

[1]IBM ILOG `CPLEX` Optimizer — High-performance mathematical programming solver for linear programming, mixed-integer programming, and quadratic programming. https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer.

The vastly dissimilar objective values listed in Table 1.1 lead to several key observations and implications. For starters, `Gurobi`[2] 2.0 is the only solver that obtained an objective value that is accurate up to at least two decimal places for both its primal and dual simplex algorithms. Interestingly, the dual algorithm of `CPLEX` 12.1 also attained this level of accuracy, but its primal algorithm provided a solution whose objective deviated by nearly 1% from the optimal value. This result is puzzling since the linear programming property of strong duality guarantees that finite primal and dual optimal objective values are equal to each other. Even more perplexingly, the respective deviations yielded by the newer `CPLEX` 12.4 runs were markedly higher than the `CPLEX` 12.1 runs due most likely to differing effects of roundoff errors in the two versions—we reran the `CPLEX` 12.1 and confirmed the previous results to rule out differences in computing environments and/or solver settings from the previous run. This result seems to imply that algorithmic advances may actually exacerbate the effects of roundoff errors on certain problems. The most concerning observation is that various other solvers yielded even higher objective value deviations, the highest of which is tantamount to being 3% away from optimality. It is straightforward to see that relying on an "optimal" solution with this degree of suboptimality could have undesirable consequences.

The takeaways and impact of the preceding example extend beyond the realm of linear programming. In particular, mixed-integer programs are solved via variations and extensions of the branch-and-bound algorithm, which breaks down the main problem into a series of linear programming subproblems. Thus, by employing this efficient process, mathematical programming solvers are repeatedly exposed to linear programming roundoff errors and, consequently, similarly incongruous outputs can

[2] `Gurobi` — The state-of-the-art mathematical programming solver for prescriptive analytics. http://www.gurobi.com.

occur when solving these more complex types of problems. Additionally, roundoff errors may also cause many cut generators—which often derive cuts directly from the simplex tableau—to invalidly cut off feasible solutions [56]. Hence, the accumulation of roundoff errors in linear programming can have considerable effects on the solution of mixed-integer programming problems and on the output validity of their associated solvers.

As the foregoing discussion explained, the potential for commercial mathematical programming solvers to be severely overrun by roundoff errors is more than just an academic concern. For this reason, today's practitioner needs to be aware of this menacing prospect and become armed with efficient tools that minimize it or prevent it altogether. While incongruous outputs like those summarized by Table 1.1 are rare, they can occur and, most disturbingly, they are virtually unrecognizable from the majority of acceptable outcomes encountered when using commercial optimization software. Moreover, based on mathematical programming solvers' continued reliance on linear programming to tackle more general classes of problems, such sizeable errors have widespread repercussions. In light of these observations, this dissertation is concerned with developing efficient foundational computational tools that are invulnerable to the worst-case pernicious effects of roundoff errors in linear and mixed-integer programming, although they could also be applied toward more general classes of problems. In fact, the factorization-based framework herein developed is designed to be integrated within existing state-of-the-art exact mathematical programming solvers to increase their efficiency. Thus, this dissertation seeks to make contributions that can be readily implemented to advance the current state of this field and thereby immediately help avoid the worst-case scenarios associated with roundoff errors.

1.2 Contributions and Overview of the Dissertation

This dissertation makes several contributions to the fields of numerical linear algebra, computer science, and mathematical programming. More specifically, it develops efficient factorization-based algorithmic tools for solving systems of linear equations exactly; these tools are designed to improve the speed and robustness of state-of-the-art exact mathematical programming solvers. The main contributions can be summarized as follows:

- The *Roundoff-Error-Free (REF) LU and Cholesky factorizations*, which are constructed entirely in integer arithmetic, and whose entries' bit-length (i.e., the number of bits required for expression) is bounded polynomially without the use of greatest common divisor operations.

- *REF forward and backward substitution algorithms*, which allow for the efficient and exact solution of linear systems by keeping the associated entry bit-lengths within the REF factorization bounds.

- *The push-and-swap approach* for updating the REF factorizations efficiently and without accruing roundoff errors; the novel approach relies on the introduction of *frame matrices*.

- *Special row/column addition, deletion, and replacement update algorithms* for the REF LU (and Cholesky) factorizations that keep the associated entry bit-lengths within the REF factorization bounds.

- A derivation and analysis of the rational arithmetic Crout and Doolittle LU factorizations vis-á-vis the REF LU factorization.

- An *efficient adaptation* of the Q-matrix revised simplex method for validating basic solutions of linear programs.

- *A generalization and theoretical expansions* of the integer-preserving Gaussian elimination algorithm.

The ensuing paragraphs provide an overview of the contents of this dissertation.

Section 2 opens by succinctly tracking the origin and progression of strategies for the fundamental problem of solving a system of linear equations (SLEs). Then, the discussion shifts to the more difficult problem of solving these systems exactly with a special focus on the computational drawbacks associated with performing the standard algorithms in exact rational arithmetic. Lastly, the section outlines several alternative methods for obtaining exact solutions to SLEs including integer-preserving Gaussian elimination, which is the underlying algorithm of the tools herein developed.

Section 3 begins with a brief introduction to linear programming (LP). Then it reviews different approaches for solving linear programs exactly and summarizes related computational results. The discussion includes a description of the mixed-precision approaches behind two state-of-the-art exact LP solvers. The discussion sheds light on the context under which the dissertation contributions would be prospectively implemented.

Section 4 develops new results in numerical linear algebra. In particular, we derive the roundoff-error-free (REF) Cholesky factorization and the REF LU factorization via inductive proofs. Afterward, we develop custom REF forward and backward substitution algorithms for applying the REF factorizations toward calculating exact solutions to SLEs efficiently. Every component of the presented REF factorization framework shares two key properties: (1) its constituent operations—

divisions included—preserve integrality and (2) its individual matrix entries have polynomially bounded bit-length without having to use greatest common divisor calculations. The section also explains the connecting properties between the traditional LU and Cholesky factorizations and their REF counterparts, and it analyzes the full computational complexity of all the therein featured REF tools.

Section 5 opens with a brief review of recent enhancements and computational tests of efficient techniques for solving SLEs exactly, which serves primarily to set the subsequent experimental design. The ensuing subsection establishes a direct relationship between the REF LU factorization and the exact rational arithmetic forms of two well-known LU factorizations: the Doolittle factorization and the Crout factorization. This theoretical exposition is accompanied by a set of computational tests designed to assess the computational performance and storage requirements of each of these exact SLE solution tools. In this section, we also develop an adaptation of the Q-matrix revised simplex method [7] (itself an extension of Edmond's Q-matrices [25]), which defines its efficient implementation as a basic solution validation tool for exact LP. A corresponding set of computational tests compares said adaptation to the REF factorization framework.

Section 6 continues the development of the REF factorization framework. In particular, it starts by explaining why applying the traditional delete-insert-reduce approach to update the REF factorizations can be costlier than constructing the corresponding factorizations from scratch. Hence, we develop the novel push-and-swap approach for updating the REF factorizations, which achieves the computational savings traditionally expected of factorization update algorithms while still avoiding roundoff errors. The approach prevents further bit-length growth in the factorization matrices' entries by taking the recursive relationships between adjacent rows and columns of the REF factorizations into account. The featured update operations are

column addition, deletion, and replacement; in addition, we prove that the complementary row updates can be performed via the column updates. In the process of deriving the REF factorization updates, the section also introduces a set of operations that augment the versatility of the integer-preserving Gaussian elimination algorithm and of the REF factorization framework.

Section 7 concludes the work by summarizing the main contributions herein contained.

2. PRELIMINARIES

This section describes the fundamental problem of solving a system of linear equations (SLE) as well as the more intricate related problem of obtaining its exact solution (i.e., without roundoff errors). For the ensuing discussion, it is useful to distinguish a specific class of roundoff-error-free (REF) algorithms within this context. In particular, we will classify a *REF Gaussian elimination algorithm* as one that finds an exact solution to a given full row-rank SLE with fixed dimensions and bounded entries in a predictably-exact number of steps via row-reduction operations (e.g., rational arithmetic Gaussian elimination, division-free Gaussian elimination, integer-preserving Gaussian elimination). More generally, a Gaussian elimination algorithm is one with the above properties, but which may or may not guarantee exact solutions.

A number of notational conventions are used throughout this dissertation and are hereby stated for the sake of clarity. Matrices are represented by uppercase letters (e.g., A, B, L) and vectors by lowercase boldface letters (e.g., $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{b}$). In addition, The ith row of a matrix A is denoted as $A_{(i,:)}$ while its jth column is denoted either as $A_{(:,j)}$ or as the column-vector $\boldsymbol{a}_j$. Scalars and individual matrix and vector entries with their appropriate indices are represented as normal lowercase letters.

It is also expedient to list the following base definitions and assumptions, which are at times slightly modified in different parts of this work.

2.1 Basic Definitions and Assumptions

Assumption 1. *Let $A\boldsymbol{x} = \boldsymbol{b}$ be a nonsingular system of linear equations with left-hand side (LHS) coefficient matrix $A \in \mathbb{Z}^{n\times n}$, right-hand side (RHS) vector $\boldsymbol{b} \in \mathbb{Z}^n$, and variable vector $\boldsymbol{x} \in \mathbb{Q}^n$.*

Definition 1. *For some Gaussian elimination algorithm, let A^k be the kth-iteration matrix, for integer $0 \leq k \leq n$, where $A^0 := A$. Denote the individual entries of this matrix as $a_{i,j}^k$, for $1 \leq i, j \leq n$.*

Definition 2. *Let $[A|\boldsymbol{b}]^k$ be the kth-iteration augmented matrix of a Gaussian elimination algorithm, for integer $0 \leq k \leq n$, where $[A|\boldsymbol{b}]^0 = [A|\boldsymbol{b}]$. With a slight abuse of notation, denote the individual entries of this matrix as $a_{i,j}^k$, for integers $1 \leq i \leq n$ and $1 \leq j \leq n+1$ (i.e., column index $n+1$ corresponds to the iterative RHS vector that is originally $\boldsymbol{b}$).*

Definition 3. *Let scalar ρ^k denote the pivot element selected from A^{k-1} to perform the kth iteration of some Gaussian elimination algorithm, where $\rho^0 := 1$. The coordinates of ρ^k, denoted as the ordered pair (r_k, c_k), are chosen from rows and columns unselected in previous iterations.*

Assumption 2 (temporary)**.** *Fix $\rho^k = a_{k,k}^{k-1} \neq 0$ (i.e., $r_k = c_k = k$), for $k \geq 1$. (Starting in Section 6.4, this assumption will be removed.)*

We remark that, based on this convention, it is not necessary to perform any row or column permutations between iterations of a Gaussian elimination algorithm; equivalently, each corresponding permutation matrix is the identity matrix.

Definition 4. *The bit-length of an integral matrix entry is given by the number of bits required to store it.*

Definition 5. *The bit-length of a rational matrix entry is given by sum of the bit-lengths of its integer numerator and integer denominator.*

Definition 6. *The solution size is given by the maximum bit-length required by an individual entry of the solution vector.*

2.2 Solving Systems of Linear Equations

2.2.1 A Brief History

The solution of simultaneous linear equations impacts virtually every area of human endeavor, and it is a mathematical concern dating back to antiquity. More than 2,000 years ago, the ancient Chinese recorded in the *Jiuzhang Suanshu* (*Nine Chapters of the Mathematical Art*) the first known process for solving systems of linear equations (SLEs) [77]; the ancient text covers instances of up to five equations and five unknowns. The development of this advanced process was tied to the solution of agricultural problems, and its application was facilitated by the rod numeral system [52]. Since then this algorithm has been rediscovered independently on multiple occasions, and it is known today as Gaussian elimination. Most notably, in the early 1800s, C. F. Gauss, after whom the method is named, developed a formal description of the procedure through which n linear equations with n unknowns could be methodically solved. The central ideas of this process are still very much in use today through various enhanced implementations—including, in part, the one elaborated in this dissertation.

Like most mathematical tools of the pre-World War II era, the application of Gaussian elimination was limited to relatively small problems due to the general limitations of manual arithmetic. This centuries-old obstacle was swiftly removed with the invention of the computer and, now, problems that would have taken years to solve manually can be easily handled in a matter of seconds. However, this unforeseen capability also has engendered unanticipated consequences. In particular, as Section 1 explains, in large and complex optimization problems the typical floating-point implementation of Gaussian elimination causes individual roundoff errors that may propagate and severely affect the validity of the solution obtained. Based on the

size of these problems alone, the specific source of these errors may be hard to pinpoint, oftentimes leaving ad hoc software parameter-tuning as the only viable option [49]. In light of this, there is a need for a systematic approach that guarantees exactness for every problem it tackles but which is sufficiently fast to be implemented in practice. To this end, this section describes the most prominent techniques that have been developed with this objective in mind—though they may not achieve this goal for every problem instance, with the exception of exact rational arithmetic Gaussian elimination and integer-preserving Gaussian elimination. Beforehand, however, this section outlines the Gaussian elimination step and its connection to the two prevalent direct methods for solving SLEs efficiently: the inverse matrix approach and triangular matrix factorization.

2.2.2 The Gaussian Elimination Step

The strategy of Gaussian elimination for solving $A\boldsymbol{x} = \boldsymbol{b}$ efficiently is to remove the presence of variable x_k from equations $A^{k-1}_{(i,:)}\boldsymbol{x} = b^{k-1}_i$ for $k = 1$ to n; where $1 \leq i \leq n$, such that $i \neq k$. Typically, the equation $A^{k-1}_{(k,:)}\boldsymbol{x} = b^{k-1}_k$ is also scaled so that $a^k_{k,k} = 1$, but this subsection ignores this detail for the sake of simplicity. We refer the reader to any book on elementary linear algebra or linear programming for a more comprehensive treatment of this algorithm and its popular variants (e.g., [2, 72, 78]). For future reference, the ensuing paragraphs simply give a mathematical description of one of its elimination steps according to the algorithm's traditional definition, and they explain how the algorithm was first utilized to calculate efficiently the solution of systems with the same LHS matrix but with different RHS vectors. For describing an elimination step, it is useful first to display the contents of the (k–1)th-iteration

matrix associated with this algorithm:

$$[A|\boldsymbol{b}]^{k-1} = \left[\begin{array}{cccccc|c} a_{1,1}^0 & 0 & \dots & a_{1,k}^{k-1} & \dots & a_{1,n}^{k-1} & a_{1,n+1}^{k-1} := b_1^{k-1} \\ 0 & a_{2,2}^1 & \dots & a_{2,k}^{k-1} & \dots & a_{2,n}^{k-1} & a_{2,n+1}^{k-1} := b_2^{k-1} \\ \vdots & \ddots & \ddots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & a_{k,k}^{k-1} & \dots & a_{k,n}^{k-1} & a_{k,n+1}^{k-1} := b_k^{k-1} \\ 0 & 0 & \dots & a_{k+1,k}^{k-1} & \dots & a_{k+1,n}^{k-1} & a_{k+1,n+1}^{k-1} := b_{k+1}^{k-1} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & a_{n,k}^{k-1} & \dots & a_{n,n}^{k-1} & a_{n,n+1}^{k-1} := b_n^{k-1} \end{array}\right]$$

where $a_{i,j}^0 = a_{i,j}$. According to Assumption 2, $a_{k,k}^{k-1}$ is selected as the kth pivot element and the eliminations of iteration k are performed by adding appropriate multiples of row k to rows 1 to k–1 and rows k+1 to n so that the column k elements $a_{1,k}^{k-1}, \dots, a_{k-1,k}^{k-1}, a_{k+1,k}^{k-1} \dots a_{n,k}^{k-1}$ all reduce to 0. Thus, given $[A|\boldsymbol{b}]^{k-1}$, the kth-iteration, (i,j)-entry is obtained via the formula:

$$a_{i,j}^k = a_{i,j}^{k-1} - \frac{a_{k,j}^{k-1}}{a_{k,k}^{k-1}} a_{i,k}^{k-1}, \tag{2.1}$$

for $1 \leq j \leq n$+1 and $1 \leq i, k \leq n$ such that $i \neq k$. Hence, after n–1 elimination steps, the algorithm yields the following diagonal matrix:

$$\left[\begin{array}{ccccc|c} a_{1,1}^0 & 0 & 0 & \dots & 0 & a_{1,n+1}^{n-1} \\ 0 & a_{2,2}^1 & 0 & \dots & 0 & a_{2,n+1}^{n-1} \\ 0 & 0 & a_{3,3}^2 & \ddots & 0 & a_{3,n+1}^{n-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a_{n,n}^{n-1} & a_{n,n+1}^{n-1} \end{array}\right].$$

Finally, the rows of the augmented matrix are scaled so that the LHS matrix becomes I_n, the identity matrix of order n, and the solution vector $\boldsymbol{x}$ equals the resulting RHS vector.

The original technique for efficiently calculating the solution of different RHS vectors while keeping the same LHS coefficient matrix (i.e., the solution of each RHS vector yields a distinct variable vector $\boldsymbol{x}$) involves a straightforward extension of the above algorithm. In particular, Gaussian elimination must instead be performed on $A\boldsymbol{x} = I_n$, which is a linear system with n RHS vectors; this system can be equivalently represented as the augmented matrix $[A|I_n]$. At its conclusion, this procedure yields a RHS matrix equal to the inverse matrix A^{-1}, by which the solution to $A\boldsymbol{x} = \boldsymbol{b}$ can be calculated as:

$$\boldsymbol{x} = A^{-1}\boldsymbol{b}. \tag{2.2}$$

In summary, after obtaining A^{-1} via Gaussian elimination, the solution vector can be calculated via matrix-vector multiplication for any RHS vector $\boldsymbol{b}$. Thus, this is known as the *inverse matrix approach*.

2.2.3 Triangular Matrix Factorization

In [12], André-Louis Cholesky developed an efficient method for solving a symmetric SLE with multiple RHS vectors that does not require storing the inverse matrix. It works by constructing a triangular factorization of A via which the SLE solution process is considerably simplified. Indeed, the solution of the resulting triangular system or, in other words the associated *triangular solve*, can be construed as reducing the process of solving n linear equations with n unknowns each to the simpler process of solving n successive equations with only one unknown each. Actually, the triangular solve must be performed twice, first using a lower triangular matrix,

and then using an upper triangular matrix; hence, the first part of this efficient solution process is also known as *forward substitution* (or *forward elimination*) and the second as *backward substitution*. Although the method turns out to be significantly superior to the inverse matrix approach, its discovery went largely unheralded until Alan Turing standardized its description and generalized its application to nonsymmetric SLEs in [81]. In particular, this work specified that an invertible matrix A can be factored as:

$$A = LU, \tag{2.3}$$

where L is a lower triangular matrix and U is an upper triangular matrix. Such a factorization is typically constructed by recording the pivots and intermediary matrices associated with a reduced version of Gaussian elimination in which row-reduction operations are performed only on the elements below and to the right of the pivot element at each iteration. This process requires $O(n^3)$ operations, and it reduces A to row-echelon form—i.e., yielding the upper-triangular matrix U when A is nonsingular.

Using the LU factorization of A, $A\boldsymbol{x} = \boldsymbol{b}$ can be equivalently evaluated by successively solving the following pair of triangular systems:

$$L\boldsymbol{y} = \boldsymbol{b} \qquad \text{and} \qquad U\boldsymbol{x} = \boldsymbol{y},$$

where $\boldsymbol{y} \in \mathbb{Q}^n$. Both triangular solves require $O(n^2)$ operations.

A factorization of the form given by Equation (2.3) need not be unique. However, when L is required to be unit lower-triangular, that is when $l_{1,1} = l_{2,2} = \cdots = l_{n,n} = 1$, this leads to a unique type of LU factorization known as the Doolittle Factorization.

Similarly, when U is instead required to be unit upper-triangular, this guarantees a unique LU factorization known as the Crout Factorization (see [33] for the proofs of uniqueness). The names of the factorizations are related to the respective algorithms used to construct them.

2.3 Obtaining Exact Solutions

The algorithms described in the preceding subsection are typically performed in fixed floating-point precision and, consequently, they are all subject to round-off errors. Naturally, the simplest way to make them immune to these errors is to work with unlimited-precision through exact rational arithmetic. This means each matrix coefficient and RHS vector coefficient of the SLE is stored as a numerator-denominator pair p/q such that p and q are arbitrary-length integers. Working in exact rational arithmetic also implies each arithmetic operation is performed in a way that no rounding errors can occur. Specifically, letting $a_1 = p_1/q_1$ and $a_2 = p_2/q_2$ be two rational coefficients in canonical form—i.e., the numerator and denominator associated with each coefficient are relatively prime—the typical arithmetic operations are carried out without roundoff error as follows:

$$a_1 \pm a_2 = \frac{p_1}{q_1} \pm \frac{p_2}{q_2} = \frac{p_1 \times q_2 \pm p_2 \times q_1}{q_1 \times q_2}, \tag{2.4}$$

$$a_1 \times a_2 = \frac{p_1}{q_1} \times \frac{p_2}{q_2} = \frac{p_1 \times p_2}{q_1 \times q_2}, \tag{2.5}$$

$$a_1 \div a_2 = \frac{p_1}{q_1} \times \frac{q_2}{p_2} = \frac{p_1 \times q_2}{q_1 \times p_2}. \tag{2.6}$$

A key detail is that the new numerator and denominator obtained at the conclusion of each operation must be stored as separate integers in order to preserve exactness.

Within the context of an algorithm, considerable additional computational costs are incurred *after* the evaluation of each numerator and denominator is completed

in each of the above expressions. They are the byproduct of greatest common divisor (gcd) calculations required to keep the bit-lengths of these integers as small as possible. Indeed, without the use of gcd, the bit-lengths of exact rational arithmetic Gaussian elimination algorithms would grow exponentially with the dimension of the matrix [72]. Consequently, state-of-the-art unlimited-precision libraries like the GNU GMP library (`GMP`)[1] automatically employ them to reduce and keep every rational to its canonical form [42]. To be consistent with this leading convention, this work assumes gcd operations are concomitant with exact rational arithmetic.

We remark that, in view of the need to keep operand bit-length as small as possible, the individual steps required to evaluate the $\pm$ operation without roundoff error, as specified by Expression (2.4), could be replaced as follows:

$$a_1 \pm a_2 = \frac{p_1}{q_1} \pm \frac{p_2}{q_2} = \frac{p_1 \times (\mathrm{lcm}(q_1, q_2) \div q_1) \pm p_2 \times (\mathrm{lcm}(q_1, q_2) \div q_2)}{\mathrm{lcm}(q_1, q_2)},$$

where $\mathrm{lcm}(q_1, q_2)$ denotes a lowest common multiple function with arguments q_1 and q_2. This alternative procedure generally yields smaller numerators and denominators, but it requires two divisions and an lcm operation, which can be just as costly as a gcd operation.

The recurring gcd operations to keep each rational's numerator-denominator pair in canonical form take a disproportionate amount of the run-time of any rational arithmetic SLE solution algorithm. To help illustrate the high costs of performing gcd operations, Table 2.1 lists the step-by-step application of the binary gcd algorithm, which the `GMP` library uses for operands with small bit-lengths. The binary gcd algorithm, which is attributed to Stein [76] but which may have been first described in the *Jiuzhang Suanshu* by the ancient Chinese in the 1st Century, has $O(N^2)$

[1]The GNU Multiple Precision Arithmetic Library. https://gmplib.org.

Table 2.1: Numerical example of the binary gcd algorithm

Iteration	p	q	Action
1	2343	147	$p \leftarrow p - q$
2	2196	147	$p \leftarrow p/2$
3	1098	147	$p \leftarrow p/2$
4	549	147	$p \leftarrow p - q$
5	402	147	$p \leftarrow p/2$
6	201	147	$p \leftarrow p - q$
7	54	147	$p \leftarrow p/2$
8	27	147	$q \leftarrow q - p$
9	27	120	$q \leftarrow q/2$
10	27	60	$q \leftarrow q/2$
11	27	30	$q \leftarrow q/2$
12	27	15	$p \leftarrow p - q$
13	12	15	$p \leftarrow p/2$
14	6	15	$p \leftarrow p/2$
15	3	15	$q \leftarrow q - p$
16	3	12	$q \leftarrow q/2$
17	3	6	$q \leftarrow q/2$
18	3	3	$q \leftarrow q - p$
19	3	0	$q \leftarrow q - p$

complexity, where N is the larger of the operands' bit-lengths [42]. The algorithm finds the gcd of integers p and q by iteratively calculating their difference when both are odd or by stripping factors of 2 when one of them is even, and then updating these operands according to the output of the executed operation; the process repeats until p or q is zero. If $p = 0$ ($q = 0$), then the gcd is returned as q (p) times the product of factors of 2 common to p and q (i.e., when both are even) in the individual steps of the algorithm.

In the featured example, the algorithm takes 19 iterations—herein characterized as individual subtractions or divisions by 2—to conclude that 3 should be factored from $p = 2343$ and $q = 147$ in order to obtain the canonical form of this numerator-denominator pair, 781/49. We remark that in Table 2.1 the stripping of factors of 2 was denoted as a division for clarity even though this task can be easily accomplished in a computer by shifting bits. Notwithstanding this shortcut, it is straightforward to infer that for wide-ranging purposes, the efforts spent in an individual gcd calculation

may have a relatively small payoff in terms of the magnitude of the factor they can ultimately remove from the numerator and denominator. This statement certainly holds for more advanced gcd algorithms with the same worst-case quadratic complexity used in practice for handling operands with large bit-lengths (e.g., Lehmer's gcd algorithm [54]). It is not difficult to arrive at the same conclusion even for the fastest known algorithms with subquadratic complexity of $O(N(\log N)^2 \log\log N)$ (e.g., Möller's left-to-right gcd algorithm [60] and Schönhage's half-gcd algorithm [70]), some of which are also available within the `GMP` library [42]. We refer the reader to Knuth [50] for an extended discussion and complexity analysis of special-purpose gcd algorithms.

Due to the rather unfavorable tradeoff between computational costs and accuracy offered by rational arithmetic, other alternatives have been developed for attempting to solve SLEs exactly. The remainder of this subsection summarizes in brief three tested prominent methods developed for this purpose as well as a supplementary technique for accelerating the rational reconstruction subroutines required by two of these methods. Then it describes an alternative algorithm, known as integer-preserving Gaussian elimination, which serves as a foundation for all the computational tools developed in this dissertation. It is worth mentioning that, unlike the latter algorithm, neither the first three methods nor the supplementary technique to be presented can be strictly classified as a REF Gaussian elimination algorithm, according to the definition provided at the beginning of this section.

2.3.1 Wiedemann's Black-Box Algorithm

Wiedemann [86] devised a symbolic method for solving sparse SLEs over finite fields that relies exclusively on field arithmetic and that accesses the input matrix A only as a matrix-vector multiplication oracle. Hence, it is known as a black-

box algorithm. By integrating the Berlekamp/Massey algorithm from coding theory [13, 57], the method functions as a randomized algorithm that finds with high probability a random vector $\boldsymbol{x}$ such that $A\boldsymbol{x} = \boldsymbol{b}$. More specifically, the algorithm is a purely algebraic method that obtains the exact solution to an SLE by calculating the minimum polynomial of a sequence of linearly generated matrix-vector products $\{A^i\boldsymbol{b}\}_{i=0}^{\infty} \in (K^n)^{\mathbb{N}}$, for a field K. Said polynomial gives an explicit formula for solving $A\boldsymbol{x} = \boldsymbol{b}$ since the linear equation,

$$c_0 I_n + c_1 A + c_2 A^2 + \cdots + c_m A^m = 0,$$

implies that the exact solution is given by:

$$\boldsymbol{x} = A^{-1}\boldsymbol{b} = -c_0^{-1}(c_1\boldsymbol{b} + c_2 A\boldsymbol{b} + \cdots + c_m A^{m-1}\boldsymbol{b}),$$

where $c_0, c_1, c_2, \ldots, c_m \in K$ [17]. The randomized portion of this process involves the choice of a row vector $\boldsymbol{u}$ that is used to generate the set of field elements $\{a_0, a_1, \ldots, a_{2m-1}\}$ with individual member a_i defined as the product $\boldsymbol{u}A^i\boldsymbol{b}$, for $0 \leq i \leq 2m$–1. The minimum polynomial of $\{A^i\boldsymbol{b}\}_{i=0}^{\infty}$ is obtained by finding the minimum degree linear generating polynomial of these field elements via the Berlekamp/Massey algorithm.

The complexity of Wiedemann's method comprises $O(n\log(n))$ calls to the black box for A and an extra $O(n^2\log(n)^2)$ in field arithmetic operations; it is assumed that K has at least $50n^2\log(n)$ entries. While these complexity measures appear promising, it has been observed that the method is much more sensitive to problem dimension relative to other competing approaches. In effect, larger instances can have a minimum polynomial of a higher degree, thereby requiring numerous matrix-vector

multiplications [17]. We refer the reader to [47] or [83] for a more comprehensive treatment of Wiedemann's black-box algorithm.

2.3.2 Iterative Refinement

Iterative refinement is an approach that calculates a solution to $\hat{\boldsymbol{x}} \approx \boldsymbol{x}$ to $A\boldsymbol{x} = \boldsymbol{b}$ of arbitrary precision via refinements yielded by a sequence of increasingly accurate approximations. In greater detail, the method begins by calculating an approximate solution $\hat{\boldsymbol{x}}_0$ and a residual vector $\hat{\boldsymbol{b}}_0 = \boldsymbol{b} - A\hat{\boldsymbol{x}}_0$ (i.e., the SLE solution error or violation) and by setting the best known approximate solution as $\hat{\boldsymbol{x}} \leftarrow \hat{\boldsymbol{x}}_0$. If the violation is not sufficiently small, the process is repeated, this time toward solving the related SLE $A\boldsymbol{x} = \hat{\boldsymbol{b}}_0$. The corresponding approximate solution $\hat{\boldsymbol{x}}_1$ and residual vector $\hat{\boldsymbol{b}}_1 = \boldsymbol{b}_0 - A\hat{\boldsymbol{x}}_1$ are utilized to refine the best known approximate solution to the original problem as $\hat{\boldsymbol{x}} \leftarrow \hat{\boldsymbol{x}} + \boldsymbol{x}_1$ and to determine if yet another similar iteration is needed. This process of solution and refinement continues until the violations are sufficiently small, that is, until $\hat{\boldsymbol{b}} < \epsilon\mathbf{1}$, where $\mathbf{1}$ denotes the n-length one-vector and where $\epsilon > 0$ is a small pre-specified precision tolerance constant.

Iterative refinement, which was first developed by Wilkinson [87] within the context of solving SLEs, was not initially intended to provide exact solutions in and of itself. That is why Ursic and Patarra [82] paired it with rational reconstruction with the idea of converting arbitrary-precision approximations into exact solutions. The rationale for being able to carry out this conversion comes from the theory of diophantine approximation and the following key theorem taken from [72]:

Theorem 2.3.1. *There exists a polynomial algorithm which, for $\alpha \in \mathbb{Q}$ and $M \in \mathbb{N}$, tests if there exists a rational number p/q with $1 \leq q \leq M$ and $|\alpha - p/q| < 1/2M^2$, and if so, finds this (unique) rational number.*

Said algorithm is applied componentwise on $\hat{\boldsymbol{x}}$ to recover $\boldsymbol{x}$ when the exact de-

nominators of each entry are small, and it works by finding the continued fraction convergent for α via the extended Euclidian algorithm (see [50]). We remark that the above result has implications beyond iterative refinement as it can be similarly implemented on the solutions obtained by other approximation approaches.

Working exclusively with integral SLEs, Wan [84] improved Ursic and Patarra's [82] hybrid strategy by modifying the iterative refinement phase into the following three step process: (1) obtain an initial approximate solution, (2) amplify this approximation by a scalar, and (3) adjust the amplified solution and corresponding residual to integers. The refinement process must be repeated until the desired precision tolerance is reached, after which rational reconstruction is performed to obtain the exact solution to the SLE. A chief reason for this algorithm's performance edge over previous versions of exact SLE iterative refinement is that, by converting floating-point approximate solutions to be integral, the algorithm maintains a scaled measure of the exact error without using extended precision.

In general, iterative refinement achieves very competitive run-times. However, since it performs nearly every computation in floating point in order to maximize speed, it is rather sensitive to numerically difficulties, which at times can cause it to fail [17, 82]. As a result, it cannot guarantee that the exact solution to a nonsingular SLE will always be found.

2.3.3 P-Adic Lifting

Dixon [21] introduced a roundoff-error-free method for solving $A\boldsymbol{x} = \boldsymbol{b}$ that relies on congruence modulo techniques, where $A \in \mathbb{Z}^{n\times n}$ and $\boldsymbol{b} \in \mathbb{Z}^n$. It consists of three main steps: (1) calculating $A^{-1}(\text{mod } p)$, for some prime p; (2) successively refining the vector $\hat{\boldsymbol{x}} \in \mathbb{Z}$ such that $A\hat{\boldsymbol{x}} \equiv \boldsymbol{b}(\text{mod } p^m)$ for a sufficiently large $m \in \mathbb{Z}$; and (3) reconstructing $\boldsymbol{x}$ from the p-adic approximation $\hat{\boldsymbol{x}}$. An important requirement for

choosing p in the first step is that it must not be a factor of $\det(A) \neq 0$. However, it is not advisable to calculate the determinant a priori; instead, different primes can be guessed since an invalid choice can be recognized in the middle of a run. Akin to the linear programming iterative refinement method, to described in Section 3.3.4, the second step of this method effectively builds an extended-precision solution iteratively. Moreover, the third step of this method relies on a modular arithmetic version of rational reconstruction; its use is justified on a result by von zur Gathen and Gerhard [83] that is analogous to Theorem 2.3.1.

The major advantage offered by the p-adic approach is that it works exclusively in integer arithmetic while performing most of the calculations in standard fixed precision. The method can be implemented in $O(n^3(\log n)^2)$ run-time, and the authors go as far as to claim that for practical cases the run-time is proportional to n^3, thus making it very efficient in most cases. However, this competitive performance has been shown to be associated with simpler problems and to degrade considerably as the bit-lengths of the denominators in the solution vector increase [17].

2.3.4 Output Sensitive Lifting

Output sensitive lifting accelerates the process of rational reconstruction and, as such, it is not a stand-alone method for solving SLEs exactly. At the same time, it is not attached to any particular method, and it can be applied to handle different types of approximate solutions. It was initially developed within the context of nonsingular SLEs by Chen and Storjohann [15], who implemented it on a p-adic lifting-based approach. Most notably, Steffy [74] expanded and bolstered the technique to improve the performances of both the iterative refinement and p-adic lifting approaches described in Sections 2.3.2 and 2.3.3, respectively. Output sensitive lifting has also been applied to compute determinants in [45] and to solve SLEs over

cyclotomic fields in [14].

The integration of the output sensitive lifting technique for solving SLEs is intended to curtail various inefficiencies associated with standard rational reconstruction. Primarily, it saves the high costs associated with using a needlessly large bound for the bit-lengths of the exact solution's denominators; in effect, the number of steps executed by the extended Euclidian algorithm and the required memory increase with the chosen bound's size. Owing to Cramer's rule, Hadamard's determinant inequality is often used to generate a valid bound, a case in point being the traditional p-adic lifting algorithm devised by Dixon [21]. However, such a bound is too pessimistic and impractical in most cases, as attested by the probabilistic analysis and computational tests performed in [1] and by experiments involving SLEs from linear programming carried out in [17]. Based on these observations, output sensitive lifting employs significantly smaller bounds, which significantly speeds up computation, and which works well when the solution denominators have small bit-lengths [15]. However, depending on the specific problem at hand, the chosen bounds may turn out to be invalid, leading to incorrect rational reconstruction output. Nonetheless, even a failed rational reconstruction run is not necessarily wasteful since the techniques developed by Steffy [74] allow for the subroutine's early termination and for using an aborted run's information to warm-start subsequent attempts. With each new rational reconstruction attempt, the employed upper bound is increased and, hence, it eventually becomes valid. This warm-start capability is also the basis for another major cost-saving strategy introduced by Steffy [74] of performing rational reconstruction at intermediate steps of an SLE solution algorithm rather than exclusively in its final step. The strategy is particularly useful for integration with iterative refinement since this algorithm increases the quality of the approximate solution at each step and since an intermediate approximation may suffice to reconstruct the

exact solution and abort execution well before the approximate-solution error falls below a specific threshold.

The enhanced performance of output sensitive lifting relative to standard rational reconstruction is dependent on solution size. In particular, this advanced technique is able to solve SLEs quickly when the solution size is small, which occurs in many applied problems. However, it is not as effective when the solution size is relatively large and, therefore, it retains the same worst-case complexity as standard rational reconstruction [74]. Moreover, its performance is also highly dependent on the quality of the approximation it receives as input. In effect, if the fixed-precision algorithm generating the approximate solution is plagued by an exceeding amount of roundoff errors, neither output sensitive lifting nor standard rational reconstruction will be able to recover the correct solution.

2.3.5 Integer-Preserving Gaussian Elimination

Integer-preserving Gaussian elimination (IPGE) is an algorithm that can be credited separately to [24], [8], [61], and even to the eponymous Gauss. As its name suggests, IPGE describes an elimination process for solving a linear system of equations $A\boldsymbol{x} = \boldsymbol{b}$, in which all operations are guaranteed to lie in the integer domain, given $A \in \mathbb{Z}^{m\times n}$ and $\boldsymbol{b} \in \mathbb{Z}^m$, for $\boldsymbol{x} \in \mathbb{Q}^n$. The final outputs of the algorithm are an integral numerator vector and an integral denominator scalar that is common to all the numerator vector entries—saving the denominator separately allows for the exact expression of solutions that may be non-integral. Since the algorithm preserves the integrality and exactness of each variable's associated numerator and denominator, the solution is exact and free of roundoff error. Hence, IPGE solves $A\boldsymbol{x} = \boldsymbol{b}$ exactly and allows its solution to be expressed up to any desired (or allowable) level of precision.

The next subsection will discuss IPGE in greater detail. For this purpose, it will be convenient to disregard the cases for which $A\boldsymbol{x} = \boldsymbol{b}$ has no solutions, which are detected in IPGE exactly like in other row-reduction algorithms (i.e., an all-zero row with a nonzero right-hand side is obtained). Hence, without loss of generality, we will assume A has full row-rank for the remainder of this section. Since adding a column to A does not change its row rank, this assumption implies the augmented matrix $[A|\boldsymbol{b}]$ formed by appending $\boldsymbol{b}$ to the right of A has full row-rank as well. Another convenient assumption we make for the remainder of this section is that the first m columns of A are linearly independent; if these columns were linearly dependent, a column-permutation matrix would need to be defined and used similarly to the row-permutation matrices discussed below in order to avoid pivot elements equal to 0.

2.3.5.1 Description of the Algorithm

IPGE works by performing integer-preserving row-reduction operations on the rows above and below the pivot element, where each resulting term is then divided by the previous pivot. Formally, given the integral full-row-rank augmented matrix $[A|\boldsymbol{b}]$, IPGE calculates the iterative entries $a_{i,j}^k$, for $k = 1$ to m, $1 \leq i \leq m$, and $1 \leq j \leq n+1$, as follows:

$$a_{i,j}^k = \begin{cases} a_{i,j}^{k-1} & \text{if } i = k \\ (\rho^k a_{i,j}^{k-1} - a_{k,j}^{k-1} a_{i,k}^{k-1})/\rho^{k-1} & \text{otherwise} \end{cases} \tag{2.7}$$

where $\rho^k = a_{k,k}^{k-1} \neq 0$, $\rho^0 = 1$, and $[A|\boldsymbol{b}]^0 = [A|\boldsymbol{b}]$. As a direct consequence of Equation (2.7), the entries of columns 1 to k of $[A|\boldsymbol{b}]^k$ can be obtained via the shortcut formula:

$$a_{i,j}^k = \begin{cases} \rho^k & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

for column $j \leq k$. Hence, the elements of the diagonal crossing the first k columns of $[A|\boldsymbol{b}^k]$, which are all equal to ρ^k, comprise the only nonzero elements among these columns.

Due to the requirement of nonzero pivots, the augmented matrix $[A|\boldsymbol{b}]^k$ is permuted row-wise prior to beginning iteration $k{+}1 \leq m$, when its $(k{+}1, k{+}1)$-entry is equal to zero. The replacement nonzero pivot, taken from rows $k{+}1$ to m of column $k{+}1$, is guaranteed to exist because $[A|\boldsymbol{b}]$ has full row-rank and from the ongoing assumption that the first m columns of A are linearly independent. Hence, after applying the shortcut given by Equation (2.8) to obtain columns 1 to k and then calculating the remaining entries via Equation (2.7), the kth-iteration matrix is updated as follows:

$$[A|\boldsymbol{b}]^k \leftarrow P^k[A|\boldsymbol{b}]^k \tag{2.9}$$

where P^k equals I_m, the identity matrix of order m, when the $(k{+}1, k{+}1)$-entry of $[A|\boldsymbol{b}]^k$ is nonzero; otherwise, P^k permutes row $k{+}1$ with a higher-index row so that $\rho^{k+1} \neq 0$.

From our ongoing assumption, it is evident that columns 1 to m of $[A|\boldsymbol{b}]^m$ form a basis B. Denoting the associated basic and nonbasic variables as $\boldsymbol{x}_B$ and $\boldsymbol{x}_N$, respectively, since $a_{1,1}^m = \ldots = a_{m,m}^m = \rho^m$ are the only nonzero elements among the

first m columns of $[A|\boldsymbol{b}]^m$, a solution to $A\boldsymbol{x} = \boldsymbol{b}$ is thus given by:

$$\boldsymbol{x}_B = \frac{\boldsymbol{b}^m}{\rho^m} \quad \text{and } \boldsymbol{x}_N = 0 \tag{2.10}$$

where $\boldsymbol{b}^m$ denotes column $n{+}1$ of $[A|\boldsymbol{b}]^m$. The following subsection will review properties of IPGE that are relevant to this work including those that explain why the above equations yield the REF solution to $A\boldsymbol{x} = \boldsymbol{b}$.

2.3.5.2 Key Properties

IPGE Property 1. The divisions in IPGE are exact.

Utilizing Sylvester's identity, Bareiss proved in [8] that the division of the expression $\rho^k a_{i,j}^{k-1} - a_{k,j}^{k-1} a_{i,k}^{k-1}$ by the previous pivot ρ^{k-1} is exact for $1 \leq i, k \leq m$ and $1 \leq j \leq n{+}1$. Based on this property, the augmented iterative matrices of IPGE are all integral and free of roundoff errors. Hence, the final solution step given by Expression (2.10) involves the division of an exact numerator by an exact denominator.

IPGE Property 2. IPGE entries have a special structure.

Edmonds [24] proved that each entry of IPGE is in fact equal to ± 1 times the determinant of a particular square submatrix of A, and is therefore integral. Specifically, $a_{i,j}^k$ (after the kth row permutation is applied to $[A|\boldsymbol{b}]^k$) may also be expressed as:

$$a_{i,j}^k = \begin{cases} (-1)^{i+k} \det\left((P^k_{\cdot} \ldots P^1 P^0 A)^{1\ldots k}_{1\ldots i-1, i+1\ldots k, j}\right) & \text{if } i \leq k \\ \det\left((P^k_{\cdot} \ldots P^1 P^0 A)^{1\ldots k, i}_{1\ldots k, j}\right) & \text{otherwise} \end{cases} \tag{2.11}$$

for $0 \leq k \leq m, 1 \leq i \leq m$, and $1 \leq j \leq n{+}1$; where $(P^k_{\cdot} \ldots P^1 P^0 A)^{1\ldots k,i}_{1\ldots k,j}$ is the submatrix induced by rows 1 to k and i and columns 1 to k and j of $P^k_{\cdot} \ldots P^1 P^0 A$ and $\det\left((P^k_{\cdot} \ldots P^1 P^0 A)^{1\ldots k,i}_{1\ldots k,j}\right)$ is its determinant. Clearly, from Equation (2.11), the

sequence of pivots ρ^1 to ρ^m is also given by the sequence of leading principal minors of $(PA)^{1\ldots m}_{1\ldots m}$ and, in particular, $\rho^m = \det\left((PA)^{1\ldots m}_{1\ldots m}\right)$. Hence, utilizing Equation (2.10), the solution to $A\boldsymbol{x} = \boldsymbol{b}$ may be restated as:

$$\boldsymbol{x}_B = \frac{\boldsymbol{b}^m}{\det\left((PA)^{1\ldots m}_{1\ldots m}\right)} \quad \text{and } \boldsymbol{x}_N = 0. \tag{2.12}$$

Together with the fact that an exact basic solution can be equivalently obtained via the equation $\boldsymbol{x}_B = [(PA)^{1\ldots m}_{1\ldots m}]^{-1}\boldsymbol{b} = \mathrm{Adj}\left((PA)^{1\ldots m}_{1\ldots m}\right)\boldsymbol{b}/\det\left((PA)^{1\ldots m}_{1\ldots m}\right)$, Equation (2.12) implies $\boldsymbol{b}^m = \mathrm{Adj}\left((PA)^{1\ldots m}_{1\ldots m}\right)\boldsymbol{b}$, where $\mathrm{Adj}((PA)^{1\ldots m}_{1\ldots m})$ is the adjunct matrix of $(PA)^{1\ldots m}_{1\ldots m}$. Consequently, since all these quantities are exact, IPGE will yield a solution that is accurate up to any desired level of precision. The remaining properties and discussion require the following definitions.

Definition 7. *Let σ and $\tilde{B}$ be the individual entry and subdeterminant, respectively, in $[A|\boldsymbol{b}]$ with the largest magnitudes, that is:*

$$\sigma = \max_{i,j} |a^0_{i,j}| \qquad \textit{and} \qquad \tilde{B} = \underset{([A|\boldsymbol{b}]^0)^R_C}{\mathrm{argmax}} \left|\det([A|\boldsymbol{b}]^0)^R_C\right|,$$

where $([A|\boldsymbol{b}]^0)^R_C$ is the submatrix induced by the rows and columns of $[A|\boldsymbol{b}]^0$ indexed by $R \subseteq \{1\ldots m\}$ and $C \subseteq \{1\ldots n\}$, respectively, such that $|R| = |C|$.

Definition 8. *Let $\omega_{\max}$ be the maximum bit-length (i.e., the maximum number of bits) required to store an individual entry of IPGE when it is applied to $[A|\boldsymbol{b}] \in \mathbb{Z}^{m\times(n+1)}$, with $m \leq n$ and initial elements with values lying in the interval $[-\sigma, \sigma]$. In more formal terms, $\omega_{\max}$ is given by the expression:*

$$\omega_{\max} = \max_{i,j,k} \lceil \log |a^k_{i,j}| \rceil.$$

IPGE Property 3. The maximum bit-length of an IPGE entry, denoted as $\omega_{\max}$, is polynomially bounded.

Based on the special structure of IPGE's entries, Hadamard's inequality [43] can be applied to obtain upper bounds on $\omega_{\max}$ as follows [9]:

$$\omega_{\max} \leq \lceil \log(|\det(\tilde{B})|) \rceil \leq \lceil \log(\prod_{h=1}^{\tilde{m}} \|\tilde{B}_{(h,:)}\|) \rceil \tag{2.13}$$

$$\leq \lceil \log(\sigma^m m^{\frac{m}{2}}) \rceil = \lceil m \log(\sigma\sqrt{m}) \rceil \tag{2.14}$$

where $\|\tilde{B}_{(h,:)}\|$ denotes the Euclidian norm of row h of $\tilde{B}$; and where $\tilde{m}$ is fixed as the number of rows of $\tilde{B}$. Notice that the bound given by Expression (2.14) does not depend on the number of columns in A.

The upper bound given by Inequality (2.13) is tight if and only if the rows (or columns) of $\tilde{B}$ are pairwise orthogonal, and it pessimistic otherwise. In particular, Abbott and Mulders [1] derived the expected value and variance of the bit-length of a determinant to be proportional to m and $\log\sqrt{m}$, respectively, for the matrix $B \in \mathbb{Z}^{m\times m}$ generated randomly as follows: choose a large m number of points on the surface of the unit sphere independently and uniformly, scale the points by an arbitrary power of ten (round/truncate for integrality), and set entry (i, j) of B as the jth coordinate of the ith point generated, for $1 \leq i, j \leq m$. Accordingly, by Inequality (2.13), the expectation of the IPGE bit-length over this type of random matrices is $O(m)$ with a variance of $O(\log\sqrt{m})$.

The upper bound given by Inequality (2.14) pinpoints the overwhelming advantage of implementing IPGE over both division-free Gaussian elimination—i.e., elimination by cross-multiplication in which, in the kth step, $a_{i,j}$ is replaced by $a_{k,k}a_{i,j} - a_{k,j}a_{i,k}$ [72]—and rational-arithmetic Gaussian elimination without gcd calculations. Specifically, the maximum entry bit-length of IPGE is bounded polynomi-

ally while that of the other two algorithms is bounded exponentially [30]. The version of rational arithmetic Gaussian elimination that performs recurrent gcd calculations also achieves the IPGE upper bounds (i.e., Inequalities (2.13) and (2.14)) for the maximum bit-lengths of each of its matrix entries' numerator and denominator [72]. Nevertheless, IPGE remains the superior alternative because gcd calculations substantially increase computational complexity and because rational arithmetic Gaussian elimination with gcd requires up to twice the storage of IPGE (i.e., IPGE's denominators are all equal to one and, consequently, do not need to be stored).

3. REVIEW OF EXACT LINEAR PROGRAMMING

This section reviews the different approaches for solving linear programs (LPs) exactly, and it summarizes certain related computational results. The discussion tracks the evolution of state-of-the art implementations, which helps set the stage for many of the algorithms developed in this dissertation. In addition, it helps explain some of the main contexts in which the proposed algorithmic tool set would be implemented and lastly how this work aims to make significant contributions to exact linear programming, or exact LP, for short. Although the ongoing narrative assumes a general familiarity with the field of linear programming, the opening subsection goes over some of its most fundamental elements.

3.1 A Brief Overview of Linear Programming

Let $A \in \mathbb{Q}^{m \times n}$ be a coefficient matrix, and let $\boldsymbol{x} \in \mathbb{Q}^n$ and $\boldsymbol{b} \in \mathbb{Q}^m$ be a decision variable vector and a RHS vector, respectively. Here and in the rest of this section, the parameters m and n represent the number of constraints and decision variables in a given linear program, respectively. Generally speaking, an LP may be stated in *standard* form as follows:

$$\text{Minimize} \qquad \sum_{j=1}^{n} c_j x_j \tag{3.1a}$$

$$\text{Subject to} \qquad \sum_{j=1}^{n} a_{i,j} x_j = b_i \quad i = 1 \ldots m \tag{3.1b}$$

$$x_j \geq 0 \quad j = 1 \ldots n; \tag{3.1c}$$

where Expression (3.1a), which can be written in vector form as $\boldsymbol{c}^T \boldsymbol{x}$, is the linear objective function; where Equations (3.1b), which can be written in matrix-vector form as $A\boldsymbol{x} = \boldsymbol{b}$, are the equality constraints; and, where Inequalities (3.1c), which can be

written in vector form as $\boldsymbol{x} \geq \mathbf{0}$, are the non-negativity constraints on the decision variables. The above LP can be extended to describe a mixed-integer programming problem (MIP) through the addition of the following integrality requirement on a nonempty subset of the decision variables:

$$x_j \in \mathbb{Z} \qquad\qquad j \in J_Z \subseteq \{1 \ldots n\}. \tag{3.2}$$

A vector $\boldsymbol{x}_0$ satisfying Constraints (3.1b) and (3.1c) for an LP—and Constraints (3.2) as well for an MIP—is said to be a *feasible solution* of the problem; if no such $\boldsymbol{x}_0$ exists, the problem is said to be *infeasible*. When there exists a feasible solution $\boldsymbol{x}^*$ such that $\boldsymbol{c}^T\boldsymbol{x}^* \leq \boldsymbol{c}^T\boldsymbol{x}$ for all feasible $\boldsymbol{x}$, $\boldsymbol{x}^*$ is said to be an *optimal solution* of the problem. Moreover, if there always exists a feasible solution $\boldsymbol{x}'$ such that $\boldsymbol{c}^T\boldsymbol{x}' < \boldsymbol{c}^T\boldsymbol{x}$ for any feasible $\boldsymbol{x}$, the problem is said to be *unbounded*.

For every LP there exists a closely related problem, known as its *dual*, which has remarkably useful properties. The dual to LP Problem (3.1) is given succinctly as:

$$\begin{aligned} &\text{Maximize} && \boldsymbol{b}^T\boldsymbol{y} && \text{(3.3a)} \\ &\text{Subject to} && A^T\boldsymbol{y} \leq \boldsymbol{c} && \text{(3.3b)} \\ & && \boldsymbol{y} \text{ unrestricted;} && \text{(3.3c)} \end{aligned}$$

where A, $\boldsymbol{b}$, and $\boldsymbol{c}$ are the same as in the original or *primal* LP; and where $\boldsymbol{y} \in \mathbb{Q}^m$ is the corresponding vector of dual decision variables. Problems (3.1) and (3.3) are connected by the property of *weak duality*, which states that, for any feasible primal solution $\boldsymbol{x}$ and feasible dual solution $\boldsymbol{y}$, the relationship $\boldsymbol{c}^T\boldsymbol{x} \geq \boldsymbol{b}^T\boldsymbol{y}$ always holds. Moreover, by the property of *strong duality*, if either Problem (3.1) or Problem (3.3) has a finite optimal solution $\boldsymbol{x}*$ or $\boldsymbol{y}*$, respectively, then so does its counterpart, and

$\boldsymbol{c}^T\boldsymbol{x}* = \boldsymbol{b}^T\boldsymbol{y}*$; if either problem has an unbounded objective, then the other problem is infeasible. A key implication of strong duality is that establishing both primal and dual LP feasibility for the same underlying basis equates to certifying the optimality of both problems. This central fact is repeatedly exploited by the advanced exact LP approaches herein reviewed.

The most popular approaches for solving LP problems are the simplex algorithm and the interior point method, which were invented by Dantzig in [18] and by Fiacco and McCormick in [31], respectively. From a geometric standpoint, the simplex algorithm works by moving between extreme points of the feasible region, and the interior point method works by traversing its interior. In theory, current versions of the interior point method have low-degree polynomial worst-case complexity [41], while a polynomial-time version of the simplex algorithm is not yet known to exist [48]. Nevertheless, formal empirical tests have shown that the simplex algorithm requires roughly $3m/2$ iterations (i.e., moves between extreme points) and seldom more than $3m$ iterations, where m is the number of constraints. In short, the simplex algorithm is very fast in practice [72].

The computational advantage of interior point methods is their ability to solve very large and sparse problems the fastest of any LP algorithms [41]. Nonetheless, the simplex method remains the superior alternative for problems of small to medium size and, most importantly, it is most expedient when taking advantage of "warm-start" information—i.e., prior knowledge about the solution. Conversely, interior point methods are generally much less useful in this respect [65]. The latter fact makes the simplex algorithm the preferred choice for performing LP sensitivity analysis and for solving MIPs via the branch-and-bound (B&B) algorithm. In particular, since the B&B algorithm works by solving linearly-relaxed subproblems that are marginally different from the linear relaxation of their respective parent nodes, the simplex

algorithm is the fundamental tool for solving MIPs efficiently.

The subject of this dissertation is the development of foundational algorithms for the efficient roundoff-error-free solution of LPs and MIPs. Hence, the ability to profit from prior solution knowledge is pivotal and, as such, our design is tailored toward the simplex-based solution approach. In light of the above observations, this work makes no further reference to interior point methods. For more details on both of the major LP solution approaches, we refer to any advanced book on optimization (e.g., [11, 65, 72]).

3.2 Unlimited-Precision Approaches

3.2.1 The Simplex Algorithm in Rational Arithmetic

A surefire approach for solving any rational LP without accruing roundoff errors is to perform a variant of the simplex algorithm in which all operations are performed in unlimited precision. The most straightforward way to implement this is to store the coefficients $a_{i,j}$, c_j, and b_i, for $1 \leq i \leq m$ and $1 \leq j \leq n$, in Problem (3.1) (or Problem (3.3)) as exact rationals and to perform every simplex operation using exact rational arithmetic. Hence, during every iteration of the algorithm, each matrix coefficient is stored as a numerator-denominator pair p/q such that p and q are arbitrary-length integers.

The above exact LP approach has been coded and tested multiple times due to its relative simplicity. An advanced implementation was performed in [4], and it involved modifying the open source `QSopt` LP solver[1] [3] to perform every operation in rational arithmetic via the GNU-GMP (`GMP`) library [42]. The authors observe that this simple modification yields a wide variance in the solution coefficients' bit-lengths and high run-times. Thus, they conclude that this naive approach is impractical in

[1] `QSopt` Linear Programming Solver. http://www.math.uwaterloo.ca/∼bico/qsopt.

most situations. A key finding in a separate implementation in [7] is that a disproportionate fraction of the run-time associated with this type of approach is spent on gcd calculations, which are required for keeping the bit-lengths of the numerators and denominators from growing exponentially [72].

3.2.2 Q-Matrices

Azulay and Pique [7] devised an alternative to the exact-arithmetic LP solver that relies exclusively on integer arithmetic. In particular, their exact revised simplex method enhances the concept of Edmond's Q-matrices [25], which are integer-valued matrices that expand the application of IPGE (see Section 2.3.5) to LP by defining a corresponding integral simplex tableau (obtained via IPGE) and its requisite pivot operations—i.e., Q-pivots. Azulay and Pique improved the Q-matrix method by replacing its previous implementation on an $m \times (n+m+1)$ full simplex tableau associated with the linear system $A + I_m = \boldsymbol{b}$ with a slimmed down version on an $m \times 2m$ augmented matrix $[B|I_m]$, where $B \in \mathbb{Z}^{m \times m}$ is a basis of A and I_m is the identity matrix of order m. By way of this smaller Q-matrix, the simplex algorithm can be applied in its revised form using integer arithmetic, and the respective basis updates can be performed in $O(m^2)$ operations via Q-pivots.

At its core, the Q-matrix revised simplex method uses a streamlined IPGE-based version of the inverse matrix approach described in Section 2.2.2. Recall that this approach calculates and stores the inverse of a nonsingular LHS coefficient matrix, and then it utilizes it to solve any SLE associated with this matrix—i.e., to solve for any RHS vector—via matrix-vector multiplication, as indicated by Equation (2.2). The difference is that the diagonalization of the LHS of $[B|I_m]$ via IPGE constructs the *adjunct* matrix of B rather than its inverse, in the RHS of the transformed augmented matrix. The adjunct matrix, denoted as $\text{Adj}(B)$, is defined as the transpose

cofactor matrix of B and, consequently, it is an integral $m \times m$ matrix. Hence, after this initial construction, individual Q-pivots actually serve to update the adjacency matrix so that it appropriately reflects simple changes to the working basis.

As result of Cramer's Rule, the matrix-vector multiplication $\mathrm{Adj}(B)\boldsymbol{b}$ yields an integral vector $\boldsymbol{x}'$ given by the equation:

$$\boldsymbol{x}' = \mathrm{Adj}(B)\boldsymbol{b} = \det(B)B^{-1}\boldsymbol{b}.$$

This indicates that once $\mathrm{Adj}(B)$ has been obtained, the Q-matrix revised simplex method can be used to calculate the exact solution to $B\boldsymbol{x} = \boldsymbol{b}$ by first carrying out the above matrix-vector multiplication and then by including $\det(B)$ as the denominator of all of $\boldsymbol{x}'$'s entries.

Azulay and Pique [7] compared the run-times of the Q-matrix and rational arithmetic revised simplex methods using 24 `NETLIB` instances. Their method had the top performance for 21 of the instances, 8 of which decreased the corresponding rational arithmetic run-times by approximately one order of magnitude. Although this implementation is not as robust as the `QSopt` rational arithmetic modification (based on the numerical libraries used and the number of instances tested), it at least hints at the potential computational advantages of replacing rational arithmetic with integer arithmetic in exact LP subroutines.

In a subsequent section of this dissertation we develop and test an efficient adaptation of the Q-matrix revised simplex method. Specifically, the version herein derived is intended to serve as a basic solution validation subroutine rather than a full-fledged exact LP solver. Additionally, the featured implementation reduces the memory requirements for constructing $\mathrm{Adj}(B)$ by one half and its underlying code is programmed using the state-of-the-art unlimited-precision `GMP` numerical library.

The corresponding derivation, discussion, and results can be found in Section 5.4.1.

3.3 Mixed-Precision Approaches

3.3.1 A Tool for LPs in Computational Geometry

Gärtner [34] developed a mixed-precision approach tailored to LP problems in computational geometry; it is included in the open-source `CGAL`[2] linear and quadratic programming solver. Specifically, Gärtner's approach is most useful when $\min(m, n)$ is small—say, no more than 30—while $\max(m, n)$ is possibly very large. His method employs floating-point arithmetic to price the nonbasic variables—thereby determining a sequence of extreme points visited by the simplex algorithm—and then arbitrary-length integer arithmetic to validate the entering variable choice and to both store and update the basis inverse. Based on the author's presentation, it is evident that it is actually the adjunct matrix rather than the inverse that is stored and updated using Edmond's Q-matrices [24]. Hence, this approach could be characterized as a partial implementation of the then-unpublished Q-matrix revised simplex method of Azulay and Pique [7].

By the author's own admission, the small number of `NETLIB` instances tested in [34] were those most advantageous computationally to their method. As a result, even though their method beat `CPLEX` version 4.0.9 in a handful of instances, it is not possible to extrapolate these conclusions to more general types of LPs. `CPLEX` actually outperforms their method for a larger number of instances and by a much wider margin, but this is to be expected since `CPLEX` works entirely in floating-point arithmetic (and is not guaranteed to deliver an exact/correct solution). Moreover, from the showcased results and a basic understanding of IPGE—upon which Q-matrices are based—it can be deduced that their method performs better than `CPLEX`

[2] `CGAL` — Computational Geometry Algorithms Library. http://www.cgal.org.

when the basis is both dense and very small for two main reasons. The first is that `CPLEX` clearly cannot take advantage of sparsity as usual and the second is that the bit-lengths of the operands are not likely to grow beyond the native precision for small basis dimension m (recall that their bit-length upper bound is proportional to m and not n, as evinced by IPGE Property 3).

Further computational comparisons in the same work showed a path for significant improvement in exact LP by hinting at the advantages of employing a mixed-precision approach rather than an unlimited-precision approach. In their study, the unlimited-precision approach simply consists of the featured algorithm with every simplex operation carried in exact arithmetic. In particular, for the 5 problems in which the two approaches are featured (both of which rely on the integer-preserving Q-matrix framework for calculating and updating adjunct matrix of the basis), the mixed-precision approach outperforms the unlimited-precision approach by approximately one order of magnitude. Thus, this suggested that considerable benefits could be attained by switching to floating-point precision in noncritical subroutines of exact LP, possibly influencing the development of the more advanced approach discussed in the next subsection.

3.3.2 The Verify and Repair Strategy

Dhiflaoui et al. [20] developed a *verify and repair* strategy for exact LP, which is claimed to handle medium to large-scale problems efficiently. Their approach verifies whether the basis returned by a floating-point solver is primal and/or dual feasible via exact arithmetic; when the verification or *basic solution validation* process contradicts the floating-point solver's output, it invokes an exact LP solver in order to resume iterating from the current basis—i.e., to repair the solution. Thus, this approach delays the use of exact computations to perform the simplex algorithm

until the basis output by the floating-point solver is deemed to be invalid.

The main experiment in [20] involved measuring the run-times taken by the `CPLEX` LP solve and by the basic solution validation subroutine for 19 `NETLIB` instances of differing sizes. The run-times taken by the latter can be up to three orders of magnitude higher than the former. Hence, exact computation remains the bottleneck of this exact LP strategy. The authors did not formally compare their mixed-precision method to an unlimited-precision LP solver and, consequently, it is not possible to contextualize these results. Moreover, due to the relative brevity of their work and the public unavailability of its corresponding code, it is not entirely clear how the exact primal and dual solutions are calculated. In spite of the absence of these and other critical details, the verify and repair strategy was instrumental in the development of state-of-the-art exact LP solvers. This is because it was the first to suggest that exact arithmetic should be used primarily to validate the output of floating-point solvers rather than to perform individual operations within the simplex algorithm.

3.3.3 LP Precision-Boosting

Koch [51] observed that simply changing the floating-point representation from 64 to 128 bits caused the `SoPlex`[3] LP solver [88, 89] to find optimal solutions to 5 numerically difficult `NETLIB` problems. Applegate et al. [4] capitalized on this insight to enhance the method of Dhiflaoui et al. [20] with the goal of minimizing the use of unlimited-precision arithmetic. Their chief improvement vis-á-vis the verify and repair approach is that their method avoids using an exact arithmetic LP solver altogether when the basic solution validation process contradicts the floating-point solver's output. Instead, their *precision-boosting* method dynamically increases the

[3] `SoPlex` — The sequential object-oriented simplex class library. http://www.zib.de/Optimization/Software/Soplex.

working floating-point precision p in the simplex algorithm each time the final basis requires repairing, as signaled by an independent exact evaluation of its primal and dual solutions. In the first LP solve, p is set to the system's native precision (e.g., 64 bits) and with each successive call to the basis repair subroutine, the LP solver resumes iterating from the last known basis in increased precision $p \leftarrow p+32k$, where $k \geq 1$; the basic solution validation subroutine is triggered each time the floating-point solver finishes executing. Thus, in the process of solving a numerically difficult LP, the basic solution validation subroutine may be repeatedly invoked, each time using the basis information returned by a higher-precision simplex run.

Applegate et al. [4] implemented their precision-boosting method by developing the `QSopt_ex` exact LP solver[4]. To enable both the dynamic precision boosts in the LP solves and the exact computation of the primal and dual solutions, their implementation makes use of arbitrary (but fixed) floating-point numbers and of unlimited-precision rationals via the `GMP` library. The principal solution verification subroutines are based on the `QSopt` LU factorization algorithms, which are exact rational arithmetic adaptations of algorithms developed by Suhl and Suhl [80]. To be precise, the authors first attempt to avoid the exact LU factorization process by finding rational approximations of the floating-point primal and dual solutions via the output sensitive lifting technique and then testing the validity of the solver's output via these approximations. This technique, described in Section 2.3.4, is an efficient version of rational reconstruction, which is itself based on the concepts of Diaphontine approximation—i.e., finding continued fraction convergents by applying the extended Euclidean algorithm. This shortcut works well on easier LP instances whose solution denominators have shorter bit-lengths, but it is less successful on more intricate problems [15, 17, 74].

[4] `QSopt_ex` Rational LP Solver. http://www.math.uwaterloo.ca/∼bico/qsopt/ex.

The authors tested their method on numerous instances from four classes of problems encompassing small (<1,000 constraints), medium (1,000-10,000 constraints), and large-scale LPs (>10,000 constraints) as well as modest-sized MIPs—for these, `QSopt_ex` was utilized to solve each LP encountered within the B&B algorithm. A detailed report of all their computational results can be found in Espinoza [28]. Most notably, for nearly all 625 benchmark LP instances, `QSopt_ex` is able to provide an accurate solution and its accompanying certificate of optimality, infeasibility, or unboundedness with relatively few precision boosts to the floating-point solver: 51.9% with none (i.e., double precision or 64 bits), 74.0% with at most one (128 bits), 81.6% with at most two (192 bits), and 98.9% with at most three (256 bits).

The `QSopt_ex` solver was not directly compared with the naive rational arithmetic `QSopt` modification (see Section 3.2.1). However, the results of separate experiments measuring the run-time ratios of each method versus the standard floating-point `QSopt` solver suggest that the precision-boosting approach is decidedly superior. For the rational arithmetic `QSopt` modification, this ratio is greater than 100 on average and as large as 27,000 for 170 small problems; for `QSopt_ex`, the geometric mean of this ratio is less than 7 for 324 small to large problems. Nevertheless, for many problems tested in the `QSopt_ex` experiment, a disproportionate amount of run-time is still spent performing the exact rational arithmetic verification subroutines. In particular, 35% of the instances requiring at least one precision-boost in [28] spend more than 30% of the run-time working with exact rational arithmetic. Because these results include easier problems that are able to avoid the exact LU factorization altogether, this figure is likely higher for numerically difficult problems. This appears to be supported by the fact that nearly 10% of the instances with run-times exceeding 1,000 seconds spend more than 80% of the run-time working with exact rational arithmetic.

3.3.4 LP Iterative Refinement

Gleixner at al. [38] devised an approach for calculating LP solutions of arbitrary precision that extends the concept of iterative refinement for solving systems of linear equations introduced by Wilkinson [87] (see Section 2.3.2). The LP iterative refinement method works by solving a sequence of closely related LPs in fixed precision, with each successive one computing a correction of the previous problem's approximate solution, until the error falls below a stated tolerance. Specifically, given the primal-dual solution vector pair $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$, obtained via a standard floating-point LP solve, the method first calculates the primal and dual violations $\hat{\boldsymbol{b}} = \boldsymbol{b} - A\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{c}} = \boldsymbol{c} - A^T\hat{\boldsymbol{y}}$, respectively, using exact rational arithmetic. Based on these violations, a new problem is constructed by shifting and scaling the primal and dual feasible regions of the original LP; the objective function, right-hand sides, and variable bounds are transformed in this fashion to zoom in on the violation, but the constraint matrix is left unchanged. Afterward, the affiliated problem is solved, its output refines the value of $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$, and the primal and dual violations are recalculated in exact rational arithmetic using the updated solutions. This iterative process continues until the solution violations are sufficiently small—i.e., until $\hat{\boldsymbol{b}}, \hat{\boldsymbol{c}} < \epsilon\mathbf{1}$, where $\mathbf{1}$ denotes the n-length one-vector and where $\epsilon > 0$ is a small pre-specified precision tolerance constant.

LP iterative refinement takes the floating-point solver as a black-box oracle and, as such, it can even be paired with an interior point-based solver; that being said, a simplex-based solver is preferable since the LPs sequentially solved are highly similar. The convergence of the LP iterative refinement method relies on the assumption that the solver returns solutions within absolute tolerances, which is generally false since floating-point precision is relative by design [39]. Hence, although they expect

a modern LP solver to provide satisfactory results in practice, Gleixner et al. [39] explain that it may produce meaningless results for very poorly-conditioned LPs. In such problematic cases, they try re-solving with a series of different ad hoc parameter settings until successful, or the process terminates without reaching the desired tolerances. The authors also suggest, but do not test, boosting the precision of the solver in a similar fashion as `QSopt_ex` to deal with these numerically difficult cases.

Gleixner et al. [38, 39] implemented LP iterative refinement via an adaptation to the simplex-based `SoPlex` solver, and they utilized the `GMP` [42] and `EGlib`[5] [29] libraries for exact arithmetic and fast memory allocation, respectively. In all, Gleixner et al. [39] performed their experiments on 1,202 instances from several standard repositories, and their method converged successfully for 1,195 of them; for the remaining instances, the LP solver timed out or failed to return an optimal solution following multiple settings readjustments. For the successful instances that required at least one round of iterative refinement, the shifted geometric mean run-time (with shifts of two seconds and 100 simplex iterations) was only 7% higher than the standard `SoPlex` run (i.e., without iterative refinement) when the maximum violation was set to $\epsilon = 10^{-50}$ and 19% higher when it was set to $\epsilon = 10^{-250}$.

The above computational results are strong evidence that LP iterative refinement allows floating-point solvers to provide significantly better approximations at low cost. It is important to note, however, that the LP iterative refinement approach does not provide or guarantee exact solutions by itself. For this reason, Gleixner [37] tested two separate extensions via `SoPlex` through which solutions without violations can be obtained: exact basic solution validation and rational reconstruction via the advanced output sensitive lifting technique (see Section 2.3.4). Both subroutines can be expensive and, consequently, their use is minimized in each respective

[5]EGlib — Efficient General Library. http://dii.uchile.cl/∼daespino/EGlib_doc.

implementation. The basic solution validation subroutine is triggered after the solver returns the same basis for a specified number of iterative refinement rounds, while the rational reconstruction subroutine is performed at a geometric frequency so that the associated effort grows linearly [37]. The former builds the exact solution by calculating an LU factorization and performing two triangular solves in exact rational arithmetic—similar to `QSopt_ex`—and the latter takes the increasingly accurate approximate solutions yielded by iterative refinement and converts them to the exact solution using extended fractions. In greater detail, the foregoing rational reconstruction subroutine rounds the solution entries to the largest convergent in the continued fraction expansion with denominators less than a pre-calculated bound. Its efficiency is further enhanced by employing various efficient techniques including Cook and Steffy's version of DLCM (i.e., solution-denominator lowest common multiple) described in [17], which accelerates component-wise reconstruction by taking advantage of the fact that the denominators of the solution vector share common factors.

Gleixner [37] showed that basic solution validation (i.e., the exact LU factorization approach applied via exact rational arithmetic) is preferable to rational reconstruction as an exact extension to LP iterative refinement using the 1,202 aforementioned instances from [39]. In particular, the rational reconstruction implementation is 5% faster only for instances that can be solved in less than .5 seconds, whereas the exact LU factorization implementation was 46% faster over all instances. On a deeper level, the exact LU factorization implementation leads to fewer iterative refinement rounds on average, thereby making its overall performance more efficient. A separate experiment in [37] on the full set of instances also revealed that LP iterative refinement with basic solution validation is 1.85 to 3 times faster than the precision-boosting `QSopt_ex` solver . Nonetheless, there is ample room for improvement for

this exact version of LP iterative refinement. For instance, although five instances could be solved only by the exact LU factorization approach, there were still three that could be solved only by rational reconstruction. Moreover, in 17 instances the exact LU factorization subroutine took up more than 90% of the total run-time, and in 4 others the two-hour time limit was reached in mid-process.

3.4 Connections with This Dissertation

If recent research is any indication, the most promising direction for exact LP points to the continued improvement and eventual convergence of the approaches heretofore discussed. As the above-mentioned computational results suggest, for example, rational reconstruction should be used as a first measure of exactness verification and, when this initial check is unsuccessful, basic solution validation should be utilized thereafter. Additionally, although LP iterative refinement with basic solution validation is generally faster than the precision-boosting verify and repair approach, the latter was able to solve six out of seven numerically difficult instances that the former could not in the studies conducted by Gleixner [37]. Thus, it seems that LP iterative refinement could benefit from precision-boosting. More generally, the most competitive and effective exact LP implementations will be those that apply a combination of the best known techniques. Along this line of reasoning, the present work focuses on enhancing the exact factorization subroutines used by both the precision-boosting and iterative refinement exact LP approaches.

Both of the state-of-the-art exact LP approaches described in Sections 3.3.3 and 3.3.4 effectively compartmentalize the exact computations from the floating-point computations. That is to say, if the exact rational arithmetic LU factorization and the related triangular solves were replaced by another exact method for computing the primal and dual solutions, the inputs and outputs of the LP solves would be

unaffected in both cases. This compartmentalization also means that, generally speaking, significant computational gains in the exact arithmetic subroutines would result in proportional gains in the overall exact solution process, especially since exact computations are the bottleneck of mixed-precision exact LP implementations [17]. Hence, making major improvements to these subroutines, independent of their prospective implementation and interface with the simplex algorithm, is a worthwhile contribution. In light of these observations, the roundoff-error-free basic solution validation techniques herein developed represent significant advances to the theory and application of exact LP.

4. ROUNDOFF-ERROR-FREE ALGORITHMS FOR SOLVING LINEAR SYSTEMS VIA CHOLESKY AND LU FACTORIZATIONS[1]

Roundoff errors and their adverse effects within mathematical programming solvers originate largely from the floating-point computations of linear programming subroutines [4]. Because most solvers utilize LU and Cholesky factorizations to solve LPs, the algorithms employed to construct and implement these factorizations should have minimal roundoff error. Thus, in this section we present roundoff-error-free (REF) LU and Cholesky factorizations as well as REF forward and backward substitution algorithms for their implementation. To be specific, the roundoff errors these processes eliminate are calculation truncation errors; we assume there are no inherent errors in the data. Additionally, although the REF factorizations do not conform exactly to the output format of standard LU and Cholesky factorizations, their implementational requirements demonstrate they share properties similar to their non-REF counterparts and are, hence, labeled accordingly; this correspondence is explained in detail in Section 4.4.

This section makes the following contributions. To start, the current work is the first to present and define the REF Cholesky factorization, which allows symmetric positive definite linear systems to be solved exactly and efficiently. Second, it provides a REF LU factorization with a more natural structure than the fraction-free LU factorization by Zhou and Jeffrey [90]. Third, this work is the first to derive and prove valid REF forward and backward substitution algorithms for REF LU

[1]Reprinted in part by permission, Adolfo R. Escobedo and Erick Moreno-Centeno, Roundoff-error-free algorithms for solving linear systems via Cholesky and LU factorizations. *INFORMS Journal on Computing*, volume 27, number 4, November, 2015. Copyright 2015, the Institute for Operations Research and the Management Sciences, 5521 Research Park Drive, Suite 200, Catonsville, Maryland 21228 USA.

and REF Cholesky factorizations (which may also be applied to the fraction-free LU factorization), along with the maximum bit-length (i.e., the maximum number of bits) their individual entries require. Fourth, it derives computational complexity measures that explicitly take coefficient growth into account for the REF factorizations, the REF substitution algorithms, and integer-preserving Gaussian elimination (IPGE); as detailed in Section 2.3.5, IPGE is an efficient REF algorithm for solving SLEs that uses only integer arithmetic. Fifth, it proves that for the class of REF Gaussian elimination algorithms (see Section 2), there exists a family of SLEs whose individual solution coefficients require bit-lengths no smaller than the IPGE maximum entry bit-length (see IPGE Property 3). This fact demonstrates that, for this popular class of algorithms, the IPGE maximum entry bit-length is optimal when obtaining exact solutions to SLEs from said family.

This section is organized as follows. Section 4.1 introduces and proves an additional property of IPGE, which is the underlying algorithm of the featured REF factorizations. Section 4.2 presents the REF Cholesky and REF LU factorizations and proves their correctness. Section 4.3 introduces forward and backward substitution algorithms tailored to the REF factorizations and proves they are free of roundoff error. In addition, Sections 4.2 and 4.3 also demonstrate that the bit-length upper bounds of the individual entries and operands of the REF factorizations and the REF substitution algorithms, respectively, are equal asymptotically to the maximum bit-length of any entry encountered in IPGE. Section 4.4 lists the attributes of the REF factorizations and explains their correspondence with those of the standard LU and Cholesky factorizations. Section 4.5 gives corresponding computational complexity measures that account for coefficient growth for all the featured algorithms. Lastly, Section 4.6 concludes the present portion of the work and suggests related future avenues of research.

4.1 An Additional Property of IPGE

The main advantages of implemeting IPGE to solve SLEs exactly are summarized by its three properties introduced in Section 2.3.5.2. The following property further bolsters the choice of applying IPGE over other REF Gaussian elimination algorithms, which are classified as those that solve linear systems in a predictably-exact number of steps via row-reduction operations (see Section 2). In particular, Theorem 4.1.1 demonstrates that there exist invertible integral matrices of any finite dimension for which any REF Gaussian elimination algorithm requires storing entries with bit-lengths of at least $\lceil \log(|\det(\tilde{B})|) \rceil$ (i.e., the largest required by any IPGE entry), where $\det(\tilde{B})$ is the subdeterminant with the largest absolute value in $[A|\boldsymbol{b}]$. To this end, in a similar vein as the IPGE bit-length bound, $\omega_{\max}$, define $W_{\max}$ as the scalar denoting the maximum bit-length required by a matrix entry of a REF Gaussian elimination algorithm.

Theorem 4.1.1. *Let $\boldsymbol{b} \in \mathbb{Z}^n$ and $\boldsymbol{x} \in \mathbb{Q}^n$. Then, there exists at least one nonsingular matrix $A \in \mathbb{Z}^{n \times n}$ such that the maximum bit-length ($W_{\max}$) of any REF Gaussian elimination algorithm that solves $A\boldsymbol{x} = \boldsymbol{b}$ for $\boldsymbol{x}$ is bounded below by $\lceil \log(|\det(\tilde{B})|) \rceil$.*

Proof. For some positive integer $i \leq n$, let A be the matrix that is obtained by replacing the ith diagonal entry of the identity matrix of order n, I_n, with an integer $q \neq 0$ that has at least one prime factor that is relatively prime to the base of computation. Additionally, set b_i equal to a prime number $p > q$, such that $p > |q \times b_j|$ for all $j \neq i$.

By construction, since q is nonzero A is nonsingular and, in particular, $\det(A) = q \det(I_n) = q$. Therefore, by Cramer's Rule, the exact solution value of x_i in the

system $A\boldsymbol{x} = \boldsymbol{b}$ is given by:

$$x_i = \frac{\mathrm{Adj}(A)_{(i,:)}\boldsymbol{b}}{\det(A)} = \frac{p}{q} = \frac{\det(\tilde{B})}{q}$$

where $\mathrm{Adj}(A)_{(i,:)}$ denotes the ith row of the adjunct matrix of A, and where $\det(\tilde{B}) = p$ because $p > |q \times b_j|$ for all $j \neq i$. Notice that $\det(\tilde{B})$ is relatively prime to q (since $p > q$ and p is prime) and, therefore, the above fraction cannot be simplified further. Moreover, since at least one of q's prime factors is relatively prime to the base of computation, x_i has a nonterminating floating-point expansion and must be stored as a numerator-denominator pair to avoid roundoff errors. In other words, there does not exist a REF representation of x_i with a smaller bit-length than $\lceil \log |\det(\tilde{B})| \rceil$. This means that at some point during the row-reduction process, p (or a nonzero multiple thereof) will be obtained and will need to be explicitly stored in order to save the exact value of x_i, thereby implying $W_{\max} \geq p = \lceil \log |\det(\tilde{B})| \rceil$. □

Corollary 1 (IPGE Property 4). *There exists a family of matrices (i.e., SLEs) for which* $\omega_{\max} \leq W_{\max}$.

Proof. The statement follows from combining Inequality (2.13) with Theorem 4.1.1. □

We remark that, by performing the appropriate row-addition operations, numerous integral matrices can be constructed from the simple A matrix described in the proof of Theorem 4.1.1 that both retain q as the determinant of A and p as the maximum subdeterminant of $[A|\boldsymbol{b}]$. As a result, it is not trivial for a REF Gaussian elimination algorithm to detect a priori which matrices have the property that $\omega_{\max} \leq W_{\max}$. These observations, the avoided costs of gcd operations, and the polynomial upper bound on every IPGE entry, consequently, present a compelling case

for choosing IPGE over other REF Gaussian elimination algorithms.

A focal point to add to the preceding discussion is that maximum bit-length differs from *space complexity*, or total memory usage required by an algorithm. To be precise, Corollary 1 does not imply IPGE has optimal space complexity for the highlighted family of matrices, and it may in fact be the case that other Gaussian REF algorithms perform significantly better with respect to this measure. Nonetheless, our chief aim for introducing the above result is to argue for the use of IPGE on the basis of its polynomially bounded bit-length growth since, unlike some popular algorithms, it does not have to perform gcd operations to ensure this property. We refer the reader to [9] for a description of the space complexity of IPGE.

4.2 Roundoff-Error-Free Factorizations

This subsection presents the REF Cholesky and LU factorizations. It is important to note that, while the herein defined algorithms could be applied directly to matrices with rational entries, we restrict our attention to the integral domain in order to take full advantage of the exact divisibility properties of IPGE (i.e., restricting the algorithm to the integral domain avoids the need to store the denominator of each entry explicitly). Hence, we assume the input matrix A is integral. Without loss of generality, however, the ensuing REF factorizations and substitution algorithms still apply to rational input matrices since they can be transformed into integral matrices by multiplying all their entries by their lowest common denominator or by an adequate power of 10 when expressed in fractional form or in decimal form, respectively. Since LP problems are formulated using integers or rationals as coefficients, and since similar transformations would be required to derive corresponding REF factorizations via exact rational arithmetic methods, the nominal increment in computational effort this preprocessing step would entail is justifiable.

We remark that Zhou and Jeffrey [90] developed an alternative REF LU factorization (although they referred to it as the fraction-free LU factorization), which both formalized and "completed" the fraction-free quasi-factorization of Nakos et al. [63]. Due to the peculiar structure of the fraction-free LU factorization, however, when the input matrix A is symmetric positive definite (SPD), the resulting U matrix is not the transpose of the L matrix. Consequently, the fraction-free LU factorization does not directly imply a corresponding fraction-free Cholesky factorization. Hence, the upcoming paragraphs begin by introducing and proving the correctness of the REF Cholesky factorization, henceforth denoted as *REF-Ch*. Afterwards, the corresponding REF LU factorization, or *REF-LU* for short, is also presented. We contend that the featured factorization derivations are formal yet easy to understand because they involve formal step-by-step inductive constructions of the final matrix factors from the respective input matrix A.

4.2.1 The REF Cholesky Factorization

As its name suggests, the REF Cholesky factorization avoids the roundoff errors accrued by numerical Cholesky factorization algorithms. To achieve this, REF-Ch requires the input matrix A to be integral (as well as SPD), that is, $A \in \mathbb{Z}^{n \times n}$. Starting with an integral matrix enables the use of IPGE-type pivoting operations, which avoid roundoff error while polynomially bounding bit-length from above, as explained in Section 2.3.5. Before introducing REF-Ch, we state a notational choice: for a matrix $B \in \mathbb{Z}_{+}^{m \times n}$, we take $\sqrt{B}$ to be the element-wise square root operator on B (thus, $\sqrt{B} = \sqrt{B_{i,j}}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$). The Roundoff-Error-Free Cholesky Factorization of SPD matrix $A \in \mathbb{Z}^{n \times n}$ is given by:

$$A = (L\sqrt{D^{-1}})(L\sqrt{D^{-1}})^T \tag{4.1}$$

where $L \in \mathbb{Z}^{n\times n}$ is lower triangular with entries $l_{i,j} = a_{i,j}^{j-1}$; where $D \in \mathbb{Z}_{+}^{n\times n}$ is diagonal with entries $d_{i,i} = \rho^{i-1}\rho^{i}$; and where $a_{i,j}^{k}$ and $\rho^{k} \in \mathbb{Z}$ are the (i,j)-entry and the pivot element, respectively, of the kth-iteration coefficient matrix of IPGE for $0 \leq k \leq n$.

The matrix factors L and D of Equation (4.1) are free of roundoff errors because they are constructed strictly from the elements of IPGE iteration matrices. Evaluating D^{-1}, taking the square roots of its elements, and then left-multiplying $\sqrt{D^{-1}}$ by L in finite precision would yield roundoff errors. In fact, since the Cholesky factorization of a SPD matrix is unique and $L\sqrt{D^{-1}}$ is lower triangular, carrying out these operations yields the traditional Cholesky factorization of A. However, as subsequent subsections will explain, it is not necessary to perform these roundoff-error-inducing operations nor to store D in order to utilize REF-Ch to solve a system of linear equations. More specifically, when solving SLEs one does not need the explicit REF Cholesky factorization, $(L\sqrt{D^{-1}})(L\sqrt{D^{-1}})^{T}$, but only the L matrix, as we will show in Sections 4.3 and 4.4. Therefore, we refer to the L matrix as the *functional form of REF-Ch* (i.e., it is the only essential part of the explicit REF Cholesky factorization needed to solve SLEs). We note that although the upcoming proof makes use of fractions in its arguments, no fractional entries will arise during an actual construction of the functional form of REF-Ch.

Theorem 4.2.1. *Let $A \in \mathbb{Z}^{n\times n}$ be SPD. Then, A admits the factorization specified by Equation* (4.1).

Proof. We will first prove the following sequence of factorizations:

$$A = L^{r}(D^{r})^{-1}\hat{L}^{r} \qquad \text{for } r = 0 \ldots n \tag{4.2}$$

where the structures of the three RHS matrices are:

$$L^r = \left[\begin{array}{cccc|c} a^0_{1,1} & 0 & \dots & 0 & \\ a^0_{2,1} & a^1_{2,2} & \ddots & \vdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \\ a^0_{r,1} & a^1_{r,2} & \dots & a^{r-1}_{r,r} & \\ \hline a^0_{r+1,1} & a^1_{r+1,2} & \dots & a^{r-1}_{r+1,r} & \boldsymbol{I}_{n-r} \\ \vdots & \vdots & & \vdots & \\ a^0_{n,1} & a^1_{n,2} & \dots & a^{r-1}_{n,r} & \end{array}\right],$$

$$\hat{L}^r = \begin{bmatrix} a^0_{1,1} & a^0_{1,2} & \dots & a^0_{1,r+1} & \dots & a^0_{1,n} \\ 0 & a^1_{2,2} & \dots & a^1_{2,r+1} & \dots & a^1_{2,n} \\ \vdots & \ddots & \ddots & \vdots & & \vdots \\ 0 & \dots & 0 & a^r_{r+1,r+1} & \dots & a^r_{r+1,n} \\ 0 & \dots & 0 & a^r_{r+2,r+1} & \dots & a^r_{r+2,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & a^r_{n,r+1} & \dots & a^r_{n,n} \end{bmatrix}, \text{ and}$$

$$D^r = [\mathrm{diag}(\rho^0\rho^1, \rho^1\rho^2, \dots, \rho^{r-1}\rho^r, \rho^r, \rho^r, \dots, \rho^r)].$$

Hence, $L^r \in \mathbb{Z}^{n\times n}$ is lower triangular, $D^r \in \mathbb{R}^{n\times n}$ is diagonal, and $\hat{L}^r \in \mathbb{Z}^{n\times n}$. Notice the last $n-r$ elements of D^r are identical and, since A is symmetric, columns 1 to r of L^r are equal to rows 1 to r of $\hat{L}^r$ (i.e., from the symmetry of A, $a^r_{i,j} = a^r_{j,i}$ for all i, j, and r). We will prove A can be factored as in Expression (4.2) by induction on r. For this purpose, let $k \in \mathbb{Z}$ be such that $0 < k \leq n$. Additionally, for ease of presentation we denote as $\underset{[i,n]}{I_n(c)}$ the $n \times n$ identity matrix, I_n, whose rows

i to n (or, alternatively, whose columns i to n) have been multiplied by a scalar c.

Base case: $r = 0$. Notice that, based on the above matrix structures, $L^0 = I_n, D^0 = (D^0)^{-1} = I_n$, and $\hat{L}^0 = A^0 = A$. Thus, Factorization (4.2) holds trivially since $A = I_n I_n A$.

Inductive step: $r = k$. Assume Factorization (4.2) holds for $r = 0 \ldots k\text{–}1$. In particular, we have:

$$A = L^{k-1}(D^{k-1})^{-1}\hat{L}^{k-1} \tag{4.3}$$

$$\text{where } L^{k-1} = \left[\begin{array}{cccc|c} a^0_{1,1} & 0 & \ldots & 0 & \\ a^0_{2,1} & a^1_{2,2} & \ddots & \vdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \\ a^0_{k-1,1} & a^1_{k-1,2} & \ldots & a^{k-2}_{k-1,k-1} & \\ \hline a^0_{k,1} & a^1_{k,2} & \ldots & a^{k-2}_{k,k-1} & \\ \vdots & \vdots & & \vdots & \boldsymbol{I}_{n-k+1} \\ a^0_{n,1} & a^1_{n,2} & \ldots & a^{k-2}_{n,k-1} & \end{array}\right],$$

$$\hat{L}^{k-1} = \begin{bmatrix} a^0_{1,1} & a^0_{1,2} & \ldots & a^0_{1,k} & \ldots & a^0_{1,n} \\ 0 & a^1_{2,2} & \ldots & a^1_{2,k} & \ldots & a^1_{2,n} \\ \vdots & \ddots & \ddots & \vdots & & \vdots \\ 0 & \ldots & 0 & a^{k-1}_{k,k} & \ldots & a^{k-1}_{k,n} \\ 0 & \ldots & 0 & a^{k-1}_{k+1,k} & \ldots & a^{k-1}_{k+1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & a^{k-1}_{n,k} & \ldots & a^{k-1}_{n,n} \end{bmatrix}, \text{ and}$$

$$D^{k-1} = [\text{diag}(\rho^0\rho^1, \rho^1\rho^2, \ldots, \rho^{k-2}\rho^{k-1}, \rho^{k-1}, \rho^{k-1}, \ldots, \rho^{k-1})].$$

A necessary step for obtaining $\hat{L}^k$ from $\hat{L}^{k-1}$ is to turn the last $n - k$ elements of

column k into zeros. This can be accomplished by factoring $\hat{L}^{k-1}$ as follows:

$$\hat{L}^{k-1} = \left[\begin{array}{c|c|c} & 0 & \\ I_{k-1} & \vdots & 0 \\ & 0 & \\ \hline 0 \dots 0 & 1 & 0 \dots 0 \\ \hline 0 & \frac{a^{k-1}_{k+1,k}}{\rho^k} & I_{n-k} \\ & \vdots & \\ & \frac{a^{k-1}_{n,k}}{\rho^k} & \end{array}\right] \times \left[\begin{array}{ccccccc} a^0_{1,1} & a^0_{1,2} & \dots & a^0_{1,k} & a^0_{1,k+1} & \dots & a^0_{1,n} \\ 0 & a^1_{2,2} & \dots & a^1_{2,k} & a^1_{2,k+1} & \dots & a^1_{2,n} \\ \vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & a^{k-1}_{k,k} & a^{k-1}_{k,k+1} & \dots & a^{k-1}_{k,n} \\ 0 & \dots & 0 & 0 & \frac{\rho^k a^{k-1}_{k+1,k+1} - a^{k-1}_{k,k+1} a^{k-1}_{k+1,k}}{\rho^k} & \dots & \frac{\rho^k a^{k-1}_{k+1,n} - a^{k-1}_{k,n} a^{k-1}_{k+1,k}}{\rho^k} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & \frac{\rho^k a^{k-1}_{n,k+1} - a^{k-1}_{k,k+1} a^{k-1}_{n,k}}{\rho^k} & \dots & \frac{\rho^k a^{k-1}_{n,n} - a^{k-1}_{k,n} a^{k-1}_{n,k}}{\rho^k} \end{array}\right] \tag{4.4}$$

where $\rho^k = a^{k-1}_{k,k} > 0$ since $a^{k-1}_{k,k}$ is the kth leading principal minor of positive definite matrix A (see Section 2.3.5.2). Similarly, $\rho^1, \dots, \rho^{k-1}$ must be positive and, thus, no pivots equal to zero arise when factoring $\hat{L}^0, \dots, \hat{L}^{k-1}$ as in Factorization (4.4) (i.e., such a factorization does not require permutations of the rows of $\hat{L}^0, \dots, \hat{L}^{k-1}$). Denote the left-hand and right-hand factors of Factorization (4.4) as $LHF(4.4)$ and

$RHF(4.4)$, respectively. We can then factor $RHF(4.4)$ into:

$$
\underset{[k+1,n]}{I_n(\rho^k)^{-1}}
\begin{bmatrix}
a^0_{1,1} & a^0_{1,2} & \dots & a^0_{1,k} & a^0_{1,k+1} & \dots & a^0_{1,n} \\
0 & a^1_{2,2} & \dots & a^1_{2,k} & a^1_{2,k+1} & \dots & a^1_{2,n} \\
\vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\
0 & \dots & 0 & a^{k-1}_{k,k} & a^{k-1}_{k,k+1} & \dots & a^{k-1}_{k,n} \\
0 & \dots & 0 & 0 & \rho^k a^{k-1}_{k+1,k+1} - a^{k-1}_{k,k+1}a^{k-1}_{k+1,k} & \dots & \rho^k a^{k-1}_{k+1,n} - a^{k-1}_{k,n}a^{k-1}_{k+1,k} \\
\vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & \dots & 0 & 0 & \rho^k a^{k-1}_{n,k+1} - a^{k-1}_{k,k+1}a^{k-1}_{n,k} & \dots & \rho^k a^{k-1}_{n,n} - a^{k-1}_{k,n}a^{k-1}_{n,k}
\end{bmatrix}.
$$

For $i, j > k$, notice $\rho^k a^{k-1}_{i,j} - a^{k-1}_{k,j}a^{k-1}_{i,k} = \rho^{k-1}a^k_{i,j}$ by definition of the (i,j)-entry of the kth-iteration matrix of IPGE. Hence, an equivalent representation of $RHF(4.4)$ is:

$$
\begin{aligned}
RHF(4.4) &= \underset{[k+1,n]}{I_n(\rho^k)^{-1}} \underset{[k+1,n]}{I_n(\rho^{k-1})}
\begin{bmatrix}
a^0_{1,1} & a^0_{1,2} & \dots & a^0_{1,k} & \dots & a^0_{1,n} \\
0 & a^1_{2,2} & \dots & a^1_{2,k} & \dots & a^1_{2,n} \\
\vdots & \ddots & \ddots & \vdots & & \vdots \\
0 & \dots & 0 & a^k_{k+1,k+1} & \dots & a^k_{k,n} \\
0 & \dots & 0 & a^k_{k+2,k+1} & \dots & a^k_{k+1,n} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \dots & 0 & a^k_{n,k+1} & \dots & a^k_{n,n}
\end{bmatrix} && (4.5) \\
&= \underset{[k+1,n]}{I_n(\rho^k)^{-1}} \underset{[k+1,n]}{I_n(\rho^{k-1})} \ \hat{L}^k && (4.6)
\end{aligned}
$$

where Equation (4.6) follows by definition of $\hat{L}^k$. Similarly, $LHF(4.4)$ is factored as:

$$LHF(4.4) = \left[\begin{array}{c|c|c} & 0 & \\ I_{k-1} & \vdots & 0 \\ & 0 & \\ \hline 0 \ldots 0 & \rho^k & 0 \ldots 0 \\ \hline 0 & a^{k-1}_{k+1,k} & I_{n-k} \\ & \vdots & \\ & a^{k-1}_{n,k} & \end{array}\right] I_n(\rho^k)^{-1}_{[k,k]}. \tag{4.7}$$

Denote the block matrix in Equation (4.7) as $\bar{L}^k$, for short. Returning to the induction hypothesis, we now have:

$$A = L^{k-1}(D^{k-1})^{-1}\hat{L}^{k-1} = L^{k-1}(D^{k-1})^{-1}\bar{L}^k I_n(\rho^k)^{-1}_{[k,k]} I_n(\rho^k)^{-1}_{[k+1,n]} I_n(\rho^{k-1})_{[k+1,n]}\hat{L}^k \tag{4.8}$$

$$= L^{k-1}\bar{L}^k(D^{k-1})^{-1} I_n(\rho^k)^{-1}_{[k,n]} I_n(\rho^{k-1})_{[k+1,n]}\hat{L}^k \tag{4.9}$$

$$= L^k(D^k)^{-1}\hat{L}^k \tag{4.10}$$

where the second equality in Equation (4.8) is obtained by substituting for $\hat{L}^{k-1}$ with Equations (4.6) and (4.7). Equation (4.9) results from multiplying $I_n(\rho^k)^{-1}_{[k,k]}$ and $I_n(\rho^k)^{-1}_{[k+1,n]}$ and from shifting $\bar{L}^k$ to the left of $(D^{k-1})^{-1}$. The latter operation can be performed because the two matrices are commutative under multiplication: except for elements k to n of its kth column, $\bar{L}^k$ has the structure of an identity matrix, while diagonal elements k to n of diagonal matrix $(D^{k-1})^{-1}$ are identical. Lastly, one can verify that $L^{k-1}\bar{L}^k = L^k$ and $(D^{k-1})^{-1} I_n(\rho^k)^{-1}_{[k,n]} I_n(\rho^{k-1})_{[k+1,n]} = (D^k)^{-1}$ and, therefore, Equation (4.10) follows from substituting these expressions accordingly.

Thus, Factorization (4.2) holds for $r = k$. Since k, such that $0 < k \leq n$, was chosen arbitrarily, the sequence of factorizations (4.2) holds true for $r = 0 \ldots n$.

Having proved this result, the proof of correctness of REF-Ch is completed by observing $L = L^n = (\hat{L}^n)^T$ and $D = D^n$. $\square$

4.2.2 The REF LU Factorization

The REF LU factorization (REF-LU) is similar to REF-Ch, but it only requires $A \in \mathbb{Z}^{m \times n}$ to have full row rank; accordingly, its proof is analogous to that of REF-Ch. The exact mathematical expression for REF-LU is:

$$PA = LD^{-1}U \tag{4.11}$$

where P is a permutation matrix of order m; where $L \in \mathbb{Z}^{m \times m}$ is lower triangular with entries $l_{i,j} = a_{i,j}^{j-1}$ for $j \leq i$; where $D \in \mathbb{Z}^{m \times m}$ is diagonal with entries $d_{i,i} = \rho^{i-1}\rho^i$; where $U \in \mathbb{Z}^{m \times n}$ is upper trapezoidal with entries $u_{i,j} = a_{i,j}^{i-1}$ for $i \leq j$; and, where $a_{i,j}^k$ and $\rho^k \in \mathbb{Z}$ are the (i,j)-entry and the pivot element, respectively, of the kth-iteration matrix of IPGE applied to PA.

Theorem 4.2.2. *Let $A \in \mathbb{Z}^{m \times n}$ have full row rank. Then, A admits the factorization specified by Equation* (4.11)*.*

Proof. In order to prove this statement, we will first prove the following sequence of factorizations:

$$PA = L^r(D^r)^{-1}U^r \qquad \text{for } r = 0 \ldots m \tag{4.12}$$

where the structures of the three right-hand side matrices are:

$$L^r = \left[\begin{array}{cccc|c} a^0_{1,1} & 0 & \dots & 0 & \\ a^0_{2,1} & a^1_{2,2} & \ddots & \vdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \\ a^0_{r,1} & a^1_{r,2} & \dots & a^{r-1}_{r,r} & \\ \hline a^0_{r+1,1} & a^1_{r+1,2} & \dots & a^{r-1}_{r+1,r} & \boldsymbol{I}_{m-r} \\ \vdots & \vdots & & \vdots & \\ a^0_{m,1} & a^1_{m,2} & \dots & a^{r-1}_{m,r} & \end{array}\right],$$

$$U^r = \begin{bmatrix} a^0_{1,1} & a^0_{1,2} & \dots & a^0_{1,r+1} & \dots & a^0_{1,n} \\ 0 & a^1_{2,2} & \dots & a^1_{2,r+1} & \dots & a^1_{2,n} \\ \vdots & \ddots & \ddots & \vdots & & \vdots \\ 0 & \dots & 0 & a^r_{r+1,r+1} & \dots & a^r_{r+1,n} \\ 0 & \dots & 0 & a^r_{r+2,r+1} & \dots & a^r_{r+2,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & a^r_{m,r+1} & \dots & a^r_{m,n} \end{bmatrix}, \text{ and}$$

$$D^r = [\text{diag}(\rho^0\rho^1, \rho^1\rho^2, \dots, \rho^{r-1}\rho^r, \rho^r, \rho^r, \dots, \rho^r)].$$

Hence, $L^r \in \mathbb{Z}^{m\times m}$ is lower triangular, $D^r \in \mathbb{Z}^{m\times m}$ is diagonal, and $U^r \in \mathbb{Z}^{m\times n}$. Notice the last $m - r$ elements of D^r are identical.

We will prove A can be factored as in Expression (4.12) by induction on r. For this purpose, let $k \in \mathbb{Z}$ be such that $0 < k \leq m$.

Base case: $r = 0$. Notice that, based on the above matrix structures, $L^0 = I_m, D^0 = (D^0)^{-1} = I_m$, and $U^0 = A^0 = A$. Thus, Factorization (4.12) holds trivially since $PA = I_m I_m A$ with $P = I_m$.

Inductive step: $r = k$. Assume Factorization (4.12) holds for $r = 0 \ldots k$–1. In particular, we have:

$$PA = L^{k-1}(D^{k-1})^{-1}U^{k-1} \tag{4.13}$$

$$\text{where } L^{k-1} = \left[\begin{array}{cccc|c} a^0_{1,1} & 0 & \ldots & 0 & \\ a^0_{2,1} & a^1_{2,2} & \ddots & \vdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \\ a^0_{k-1,1} & a^1_{k-1,2} & \ldots & a^{k-2}_{k-1,k-1} & \\ \hline a^0_{k,1} & a^1_{k,2} & \ldots & a^{k-2}_{k,k-1} & \\ \vdots & \vdots & & \vdots & \boldsymbol{I}_{m-k+1} \\ a^0_{m,1} & a^1_{m,2} & \ldots & a^{k-2}_{m,k-1} & \end{array}\right],$$

$$U^{k-1} = \begin{bmatrix} a^0_{1,1} & a^0_{2,1} & \ldots & a^0_{1,k} & \ldots & a^0_{1,n} \\ 0 & a^1_{2,2} & \ldots & a^1_{2,k} & \ldots & a^1_{2,n} \\ \vdots & \ddots & \ddots & \vdots & & \vdots \\ 0 & \ldots & 0 & a^{k-1}_{k,k} & \ldots & a^{k-1}_{k,n} \\ 0 & \ldots & 0 & a^{k-1}_{k+1,k} & \ldots & a^{k-1}_{k+1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & a^{k-1}_{m,k} & \ldots & a^{k-1}_{m,n} \end{bmatrix} \text{,and}$$

$$D^{k-1} = [\text{diag}(\rho^0\rho^1, \rho^1\rho^2, \ldots, \rho^{k-2}\rho^{k-1}, \rho^{k-1}, \rho^{k-1}, \ldots, \rho^{k-1})].$$

A necessary step for obtaining U^k from U^{k-1} is eliminating the last $m-k$ elements

of column k. With this in mind, U^{k-1} can be factored as follows:

$$
U^{k-1} = \left[\begin{array}{c|c|c}
 & 0 & \\
I_{k-1} & \vdots & 0 \\
 & 0 & \\
\hline
0 \dots 0 & 1 & 0 \dots 0 \\
\hline
0 & \frac{a^{k-1}_{k+1,k}}{\rho^k} & I_{m-k} \\
 & \vdots & \\
 & \frac{a^{k-1}_{m,k}}{\rho^k} &
\end{array}\right]
$$

$$
\times \left[\begin{array}{ccccccc}
a^0_{1,1} & a^0_{1,2} & \dots & a^0_{1,k} & a^0_{1,k+1} & \dots & a^0_{1,n} \\
0 & a^1_{2,2} & \dots & a^1_{2,k} & a^1_{2,k+1} & \dots & a^1_{2,n} \\
\vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\
0 & \dots & 0 & a^{k-1}_{k,k} & a^{k-1}_{k,k+1} & \dots & a^{k-1}_{k,n} \\
0 & \dots & 0 & 0 & \frac{\rho^k a^{k-1}_{k+1,k+1} - a^{k-1}_{k,k+1} a^{k-1}_{k+1,k}}{\rho^k} & \dots & \frac{\rho^k a^{k-1}_{k+1,n} - a^{k-1}_{k,n} a^{k-1}_{k+1,k}}{\rho^k} \\
\vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & \dots & 0 & 0 & \frac{\rho^k a^{k-1}_{m,k+1} - a^{k-1}_{k,k+1} a^{k-1}_{m,k}}{\rho^k} & \dots & \frac{\rho^k a^{k-1}_{m,n} - a^{k-1}_{k,n} a^{k-1}_{m,k}}{\rho^k}
\end{array}\right]
$$

(4.14)

where, for the purpose of clarity, we assume $\rho^k = a^{k-1}_{k,k} \neq 0$ (when $a^{k-1}_{k,k} = 0$, A is multiplied by the appropriate permutation matrix as explained in Section 2.3.5.1). Denote the left-hand and right-hand factors of Factorization (4.14) as $LHF(4.14)$

and $RHF(4.14)$, respectively. We can factor $RHF(4.14)$ as:

$$RHF(4.14) = \underset{[k+1,m]}{I_m(\rho^k)^{-1}}$$
$$\times \begin{bmatrix} a^0_{1,1} & a^0_{1,2} & \dots & a^0_{1,k} & a^0_{1,k+1} & \dots & a^0_{1,n} \\ 0 & a^1_{2,2} & \dots & a^1_{2,k} & a^1_{2,k+1} & \dots & a^1_{2,n} \\ \vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & a^{k-1}_{k,k} & a^{k-1}_{k,k+1} & \dots & a^{k-1}_{k,n} \\ 0 & \dots & 0 & 0 & \rho^k a^{k-1}_{k+1,k+1} - a^{k-1}_{k,k+1}a^{k-1}_{k+1,k} & \dots & \rho^k a^{k-1}_{k+1,n} - a^{k-1}_{k,n}a^{k-1}_{k+1,k} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & \rho^k a^{k-1}_{m,k+1} - a^{k-1}_{k,k+1}a^{k-1}_{m,k} & \dots & \rho^k a^{k-1}_{m,n} - a^{k-1}_{k,n}a^{k-1}_{m,k} \end{bmatrix}.$$

Recall $\underset{[i,n]}{I_n(c)}$ is the $n \times n$ identity matrix, I_n, whose rows i to n have been multiplied by a scalar c. For $i, j > k$, notice $\rho^k a^{k-1}_{i,j} - a^{k-1}_{k,j}a^{k-1}_{i,k} = \rho^{k-1}a^k_{i,j}$ by definition of the (i,j)-entry of the kth-iteration IPGE matrix of PA. Hence, an equivalent representation of $RHF(4.14)$ is:

$$RHF(4.14) = \underset{[k+1,m]}{I_m(\rho^k)^{-1}} \underset{[k+1,m]}{I_m(\rho^{k-1})} \begin{bmatrix} a^0_{1,1} & a^0_{1,2} & \dots & a^0_{1,k} & \dots & a^0_{1,n} \\ 0 & a^1_{2,2} & \dots & a^1_{2,k} & \dots & a^1_{2,n} \\ \vdots & \ddots & \ddots & \vdots & & \vdots \\ 0 & \dots & 0 & a^k_{k+1,k+1} & \dots & a^k_{k,n} \\ 0 & \dots & 0 & a^k_{k+2,k+1} & \dots & a^k_{k+1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & a^k_{m,k+1} & \dots & a^k_{m,n} \end{bmatrix} \tag{4.15}$$

$$= \underset{[k+1,m]}{I_m(\rho^k)^{-1}} \underset{[k+1,m]}{I_m(\rho^{k-1})} U^k \tag{4.16}$$

where Equation (4.16) follows by definition of U^k. Similarly, $LHF(4.14)$ can be

factored into:

$$LHF(4.14) = \left[\begin{array}{c|c|c} & 0 & \\ I_{k-1} & \vdots & 0 \\ & 0 & \\ \hline 0 \ldots 0 & \rho^k & 0 \ldots 0 \\ \hline 0 & a^{k-1}_{k+1,k} & I_{m-k} \\ & \vdots & \\ & a^{k-1}_{m,k} & \end{array}\right] I_{m}(\rho^k)^{-1}_{[k,k]}. \tag{4.17}$$

Denote the block matrix in Equation (4.17) as $\bar{L}^k$, for short. Returning to the induction hypothesis, we have:

$$PA = L^{k-1}(D^{k-1})^{-1}U^{k-1} = L^{k-1}(D^{k-1})^{-1}\bar{L}^k I_{m}(\rho^k)^{-1}_{[k,k]} I_{m}(\rho^k)^{-1}_{[k+1,m]} I_{m}(\rho^{k-1})_{[k+1,m]} U^k \tag{4.18}$$

$$= L^{k-1}\bar{L}^k(D^{k-1})^{-1} I_{m}(\rho^k)^{-1}_{[k,m]} I_{m}(\rho^{k-1})_{[k+1,m]} U^k \tag{4.19}$$

$$= L^k(D^k)^{-1}U^k \tag{4.20}$$

where the second equality in Equation (4.18) is obtained by substituting for U^{k-1} with Equations (4.16) and (4.17). Equation (4.19) results from multiplying $I_{m}(\rho^k)^{-1}_{[k,k]}$ and $I_{m}(\rho^k)^{-1}_{[k+1,m]}$ and from shifting $\bar{L}^k$ to the left of $(D^{k-1})^{-1}$. We can perform the latter operation because the two matrices are commutative under multiplication: except for elements k to m of its kth column, $\bar{L}^k$ has the structure of an identity matrix, while diagonal elements k to m of diagonal matrix $(D^{k-1})^{-1}$ are identical. Lastly, one can verify $L^{k-1}\bar{L}^k = L^k$ and $(D^{k-1})^{-1} I_{m}(\rho^k)^{-1}_{[k,m]} I_{m}(\rho^{k-1})_{[k+1,m]} = (D^k)^{-1}$ and, therefore, Equation (4.20) follows from substituting these expressions accordingly. Thus, Factorization

(4.12) holds for $r = k$. Since k, such that $0 < k \leq m$, was chosen arbitrarily, the sequence of factorizations (4.12) holds true for $r = 0 \ldots m$.

Having proved this result, the proof of correctness of REF-LU is completed by observing $L = L^m$, $D = D^m$, and $U = U^m$. $\square$

Notice L and D in REF-LU are equal to their counterparts in REF-Ch, except that the elements of D can be negative in this case. Therefore, REF-Ch could be alternatively deduced from REF-LU. On the other hand, a REF Cholesky factorization cannot be directly deduced from the fraction-free LU factorization, which is the alternative REF LU factorization presented in [90]. This suggests REF-LU has a more natural structure than the fraction-free LU factorization.

To conclude this subsection, we remark that when solving SLEs one does not need the explicit REF LU factorization $LD^{-1}U$, but only the L and U matrices, as we will show in Sections 4.3 and 4.4. Therefore, we refer to the L and U matrices as the *functional form of REF-LU* (i.e., these are the only essential parts of the explicit REF LU factorization needed to solve SLEs).

4.3 Roundoff-Error-Free Forward and Backward Substitution

This subsection develops REF forward and backward substitution algorithms for the REF LU factorization of a matrix with full row-rank. In addition, it derives the maximum bit-length of each algorithm's individual entries. The REF substitution algorithms (and the derived bit-length bounds) also extend to the REF Cholesky factorization of a SPD matrix $A \in \mathbb{Z}^{n \times n}$ due to the following equivalent output formats:

$$A = (L\sqrt{D^{-1}})(L\sqrt{D^{-1}}) = LD^{-1}L^T = LD^{-1}U$$

where the right-most equality results from the fact that U is identical to L^T when the input matrix is a symmetric square matrix. Having explained this, let $\boldsymbol{x} \in \mathbb{Q}^n$,

$\boldsymbol{b} \in \mathbb{Z}^m$, and $A \in \mathbb{Z}^{m \times n}$ for the remainder of this subsection.

We remark that Zhou and Jeffrey [90] described forward and backward substitution algorithms for their fraction-free LU factorization. Careful inspection reveals, however, that their forward substitution algorithm and proof are incorrect. Additionally, although their backward substitution process does preserve integrality, this fact does not follow from the reasoning in its proof. Hence, the REF substitution algorithms and corresponding proofs herein presented fill significant gaps in the existing literature. An extended discussion regarding the invalidity of the fraction-free substitution algorithms is presented at the end of this subsection.

4.3.1 REF Forward Substitution

As described in Section 4.2.2, A can be expressed by its REF LU factorization (Factorization (4.11)), a roundoff-error free factorization of the form $PA = LD^{-1}U$, where P is a permutation matrix, where L and D have the same structure as the lower triangular and diagonal matrices, respectively, of the REF Cholesky factorization (except that the entries of D can be negative in this case), and where $U \in \mathbb{Z}^{m \times n}$ is upper trapezoidal. To enhance clarity, throughout this section we assume no pivot elements equal to 0 arise in the application of IPGE on A. From this assumption, we have $P = I_m$ and, thus, $A = LD^{-1}U$. Additionally, this implies columns 1 to m of A form a basis, which we henceforth denote as B.

A critical point of information to know before proceeding is that, in order to use REF-LU to solve the linear system $A\boldsymbol{x} = \boldsymbol{b}$, the REF substitution algorithms herein described must be applied to the scaled linear system $A\det(B)\boldsymbol{x} = \det(B)\boldsymbol{b}$. Section 4.3.2 will discuss the reason for working with this scaled linear system rather than the original system. As a form of shorthand, we will denote this scaled linear system as $A\boldsymbol{x}' = \boldsymbol{b}'$, where $\boldsymbol{x}' = \det(B)\boldsymbol{x}$ and $\boldsymbol{b}' = \det(B)\boldsymbol{b}$.

Using REF-LU, the system $A\boldsymbol{x}' = \boldsymbol{b}'$ can be expressed equivalently as $LD^{-1}U\boldsymbol{x}' = \boldsymbol{b}'$. The forward substitution procedures for $(m \times 1)$-vector $\boldsymbol{z}$ in $L\boldsymbol{z} = \boldsymbol{b}'$ and for $(m \times 1)$-vector $\boldsymbol{y}'$ in $D^{-1}\boldsymbol{y}' = \boldsymbol{z}$, can be combined into a single forward substitution for $\boldsymbol{y}'$ in $LD^{-1}\boldsymbol{y}' = \boldsymbol{b}'$. Equivalently, one can calculate $\boldsymbol{y}' = \det(B)\boldsymbol{y}$, where $(m \times 1)$-vector $\boldsymbol{y}$ is obtained by standard forward substitution from the equation $LD^{-1}\boldsymbol{y} = \boldsymbol{b}$ or, stated in algorithmic form:

$$y_i = \frac{d_{i,i}}{l_{i,i}}\left(b_i - \sum_{j=1}^{i-1}\frac{l_{i,j}}{d_{j,j}}y_j\right) = l_{i-1,i-1}\left(b_i - \sum_{j=1}^{i-1}\frac{l_{i,j}}{l_{j-1,j-1}l_{j,j}}y_j\right) \text{ for } i = 1 \ldots m \quad (4.21)$$

where the second equality is obtained from the fact $d_{i,i} = \rho^{i-1}\rho^i = a_{i-1,i-1}^{i-2}a_{i,i}^{i-1} = l_{i-1,i-1}l_{i,i}$ (notice the division outside the parenthesis is exact), and where we define $l_{0,0}$=1. As stated, however, this algorithm is not REF since the division in each of the above individual summands is not exact. Provided that the exact value of $\boldsymbol{y}$ is integral (which we will prove), one possible technique to make Equation (4.21) REF is to multiply each summand's numerator within the expression for y_i appropriately so that all the sum's terms share the common denominator $l_{0,0}l_{1,1}\ldots l_{i-1,i-1}$ (i.e., multiply the first summand by $l_{2,2}l_{3,3}\ldots l_{i-1,i-1}$, the second by $l_{0,0}l_{3,3}l_{4,4}\ldots l_{i-1,i-1}$, etc.). The maximum bit-length upper bound of the common denominator (and that of each numerator), however, grows to a factor of m-times the maximum bit-length of IPGE, that is, to $O(m^2\log(\sigma\sqrt{m}))$, since each entry $l_{k,k} = a_{k,k}^{k-1}$ has bit-length $\lceil k\log(\sigma\sqrt{k})\rceil$ for $0 < k < i$. In the remainder of this subsection we develop a better REF algorithm that requires only the bit-length associated with IPGE.

To derive the more efficient REF forward substitution algorithm, we define a recursive relation for solving Equation (4.21), and we prove a key property associated with the recursion. These steps then provide the insight for how to modify the operations of the standard forward substitution algorithm to obtain an equivalent

yet REF algorithm.

To develop a recursive definition for y_i, Equation (4.21) is rewritten as:

$$y_i = l_{i-1,i-1}\left(\ldots\left(\left((b_i) - \frac{l_{i,1}}{l_{0,0}l_{1,1}}y_1\right) - \frac{l_{i,2}}{l_{1,1}l_{2,2}}y_2\right) - \ldots - \frac{l_{i,i-1}}{l_{i-2,i-2}l_{i-1,i-1}}y_{i-1}\right)$$

where $i = 1 \ldots m$; and where the parentheses are placed to define the specific order in which each operation is performed. Define a triangular array of intermediary calculations $\Upsilon = \upsilon_{i,r}$, for $0 \leq r < i \leq m$, whose ith row corresponds to the operations inside the parentheses of the ordered version of Equation (4.21), performed to calculate y_i as follows:

$$\upsilon_{i,r} = \begin{cases} b_i & \text{if } r = 0 \\ \upsilon_{i,r-1} - \frac{l_{i,r}}{l_{r-1,r-1}l_{r,r}}y_r & \text{if } 0 < r < i. \end{cases} \tag{4.22}$$

Based on this recursion, $y_i = l_{i-1,i-1}\upsilon_{i,i-1}$ for all i and, thus, $l_{r-1,r-1}\upsilon_{r,r-1}$ can be equivalently inserted in place of y_r in the above expression. Interestingly, each of these recursive terms has the property that it is related to the calculation of an IPGE entry associated with $\boldsymbol{b}$, as the following lemma will demonstrate.

Lemma 1. *The recursion Υ can be defined equivalently as $\upsilon_{i,r} = a^r_{i,n+1}/l_{r,r}$ for $0 \leq r < i \leq m$; where $a^r_{i,n+1}$ is an entry of IPGE corresponding to the rth IPGE iteration on the ith component of $\boldsymbol{b}$ (recall, per definition of IPGE, that column $n+1$ corresponds to the right-hand side of the augmented IPGE iterative matrices).*

Proof. Based on Expression (4.22), $\upsilon_{i,r}$ is calculated recursively by ascending index i (i.e., since $y_r = l_{r-1,r-1}\upsilon_{r,r-1}$) and then by ascending index r. Let t be a one-dimensional index for Υ denoting the order in which $\upsilon_{i,r}$ is calculated according to

the recursive definition (e.g., $\upsilon_{t=1} = \upsilon_{1,0}, \upsilon_{t=2} = \upsilon_{2,0}, \upsilon_{t=3} = \upsilon_{2,1}$, etc). Then, since Υ is a triangular array, $t = \sum_{j=1}^{i-1} j + r + 1 = \frac{i^2-i}{2} + r + 1$ and, in particular, the elements along the diagonal of Υ are $\upsilon_{i,i-1} = \upsilon_{(i^2+i)/2}$ for $1 \leq i \leq m$. Using this alternative indexing, we prove the theorem by induction on t. For this purpose, let $k \in \mathbb{Z}$ be such that $1 < k \leq (m^2+m)/2$ (i.e. k goes from 2 up to the total number of intermediary calculations).

Base case: $t = 1$. From the definition of Υ and the entries of IPGE:

$$\upsilon_1 = \upsilon_{1,0} = b_1^0 = a_{1,n+1}^0 = \frac{a_{1,n+1}^0}{l_{0,0}}$$

where the final equation results from the fact $l_{0,0} = 1$. Thus, the lemma holds for $t = 1$.

Inductive step: $t = k$. Assume the statement holds for $t = 1 \ldots k$–1. The proof of the inductive step is divided into two cases: (1) index k–1 corresponds to a diagonal element of Υ, or (2) index k–1 corresponds to a non-diagonal element of Υ.

Case (1): $k - 1 = (i^2 + i)/2$, where $0 < i < m$. The result is similar to the base case, since:

$$\upsilon_k = \upsilon_{i+1,0} = b_{i+1} = \frac{a_{i+1,n}^0}{l_{0,0}}.$$

Case (2): $k - 1 = (i^2 - i)/2 + r$, where $0 < r < i$. Evaluating υ_k gives:

$$\upsilon_k = \upsilon_{i,r} \tag{4.23}$$

$$= \upsilon_{i,r-1} - \frac{l_{i,r}}{l_{r-1,r-1} l_{r,r}} (l_{r-1,r-1} \upsilon_{r,r-1}) = \frac{a_{i,n+1}^{r-1}}{l_{r-1,r-1}} - \frac{l_{i,r}}{l_{r-1,r-1} l_{r,r}} \left(\frac{l_{r-1,r-1} a_{r,n+1}^{r-1}}{l_{r-1,r-1}} \right) \tag{4.24}$$

$$= \left(\frac{l_{r,r} a_{i,n+1}^{r-1} - l_{i,r} a_{r,n+1}^{r-1}}{l_{r-1,r-1}} \right) \frac{1}{l_{r,r}} = \left(\frac{a_{r,r}^{r-1} a_{i,n+1}^{r-1} - a_{i,r}^{r-1} a_{r,n+1}^{r-1}}{a_{r-1,r-1}^{r-2}} \right) \frac{1}{l_{r,r}} = \frac{a_{i,n+1}^{r}}{l_{r,r}} \tag{4.25}$$

where in Expression (4.24) the first equality results from the definition of t, the second

from the definition of Υ with $l_{r-1,r-1}\upsilon_{r,r-1}$ substituting for y_r, and the third from the inductive hypothesis; and where in Expression (4.25) the first equality follows from simple algebra, the second substitutes the elements of L inside the parenthesis with the corresponding IPGE entries, and the third applies the definition of $a^r_{i,n+1}$ from IPGE. Thus, the lemma holds for $t = k$. Since k, such that $1 < k \leq (m^2 + m)/2$, was chosen arbitrarily, the result holds true for all t. □

Utilizing the succinct definition of Υ stated in Lemma 1, the recursive relation between the individual intermediary calculations is not explicit but is instead implied since $a^r_{i,n+1}$ depends on the terms $a^{r-1}_{i,n+1} = l_{r-1,r-1}\upsilon_{i,r-1}$ and $a^{r-1}_{r,n+1} = l_{r-1,r-1}\upsilon_{r,r-1}$ (notice that neither of these IPGE entries was calculated during the construction of the REF factorization). In particular, from the definition of IPGE:

$$a^r_{i,n+1} = \frac{\rho^r a^{r-1}_{i,n+1} - a^{r-1}_{i,r} a^{r-1}_{r,n+1}}{\rho^{r-1}} = \frac{l_{r,r} a^{r-1}_{i,n+1} - l_{i,r} a^{r-1}_{r,n+1}}{l_{r-1,r-1}} \tag{4.26}$$

where the second equality follows from the fact $\rho^r = a^{r-1}_{r,r} = l_{r,r}$ and $a^{r-1}_{i,r} = l_{i,r}$ for $1 \leq r \leq i \leq m$. According to the properties of IPGE (see Section 2.3.5.2), the division by $l_{r-1,r-1}$ in Equation (4.26) is exact and, therefore, the equation is REF. Next, in order to obtain a similar recursion to Equation (4.26) using only L's entries, we transform Expression (4.22) by getting rid of the division by $l_{r,r}$, which is inexact unlike the division by $l_{r-1,r-1}$ in Equation (4.26). For this purpose, define a triangular array of intermediary calculations $\Psi = \psi_{i,r} = l_{r,r}\upsilon_{i,r}$, for $0 \leq r < i \leq m$, as follows:

$$\psi_{i,r} = l_{r,r}\upsilon_{i,r} = \begin{cases} l_{0,0} b_i & \text{if } r = 0 \\ l_{r,r}\upsilon_{i,r-1} - \frac{l_{i,r}}{l_{r-1,r-1}}(l_{r-1,r-1}\upsilon_{r,r-1}) & \text{if } 0 < r < i \end{cases} \quad \text{for} \quad i = 1 \ldots m$$

where y_r of Expression (4.22) was substituted by $l_{r-1,r-1}\upsilon_{r,r-1}$. Furthermore, since $l_{0,0} = 1$, $\psi_{i,r-1} = l_{r-1,r-1}\upsilon_{i,r-1}$, and $\psi_{r,r-1} = l_{r-1,r-1}\upsilon_{r,r-1}$, this recursion can be restated

as:

$$\psi_{i,r} = \begin{cases} b_i & \text{if } r = 0 \\ \frac{l_{r,r}\psi_{i,r-1} - l_{i,r}\psi_{r,r-1}}{l_{r-1,r-1}} & \text{if } 0 < r < i \end{cases} \qquad \text{for} \quad i = 1 \ldots m. \tag{4.27}$$

Theorem 4.3.1. *The algorithm specified by Expression* (4.27) *evaluates Equation* (4.21) *without accruing roundoff errors.*

Proof. We have that $\psi_{i,i-1} = l_{i-1,i-1}v_{i,i-1} = y_i$ for all i and, thus, solving Expression (4.27) is equivalent to solving Equation (4.21). Additionally, from Lemma 1, $\psi_{i,r-1} = a^{r-1}_{i,n+1}$, $\psi_{r,r-1} = a^{r-1}_{r,n+1}$, and $\psi_{i,r} = a^{r}_{i,n+1}$, which proves Expression (4.27) equals REF Equation (4.26) for $0 < r < i$. Therefore, this proves that the division by $l_{r-1,r-1}$ in Expression (4.27) is exact and, since $\boldsymbol{b}$ is integral, that all the calculations therein are REF. □

We emphasize that the output vector $\boldsymbol{y}$ resulting from REF forward substitution in the equation $LD^{-1}\boldsymbol{y} = \boldsymbol{b}$ is obtained from the recursion given by Expression (4.27) as follows:

$$y_i = \psi_{i,i-1} \qquad \text{for} \quad i = 1 \ldots m.$$

Moreover, based on the preceding derivation, y_i is exactly the IPGE entry $a^{i-1}_{i,n+1}$, for $1 \leq i \leq m$, which was not calculated during the construction of REF-LU.

Corollary 2. *The REF forward substitution algorithm (i.e., Expression* (4.27)*) requires the same bit-length as IPGE.*

Proof. As the proof of Theorem 4.3.1 demonstrates, the calculation of every term of Ψ corresponds to a new IPGE operation on column $n+1$ (i.e., the right-hand side) of the augmented IPGE iterative matrices. Consequently, the bit-lengths of REF forward substitution and IPGE are bounded equally. □

4.3.2 REF Backward Substitution

Given $\boldsymbol{y} \in \mathbb{Z}^m$ from the REF forward substitution process described in the previous subsection, it is easy to see that traditional backward substitution for $\boldsymbol{x}$ in $U\boldsymbol{x} = \boldsymbol{y}$ will accumulate roundoff errors since $x_m \notin \mathbb{Z}$ implies all subsequent operations of the substitution will be floating-point operations. One can keep all operations in the integral domain utilizing pseudo division or rational arithmetic and then delay division until the substitution process is finished, but this is problematic given the rapid growth of the integers needed. Thus, here we devise a better alternative based on IPGE. For this purpose, recall that the indices of the basic variables associated with basis B of A, denoted as $\boldsymbol{x}_B$, are 1 to m (see Section 4.3.1). As described in Section 2.3.5, $A\boldsymbol{x} = \boldsymbol{b}$ can be solved without roundoff errors by performing m IPGE iterations on the augmented matrix $[A|\boldsymbol{b}]$, setting all nonbasic variables to 0, and then carrying out the operations:

$$x_i = \frac{b_i^m}{a_{i,i}^m} = \frac{b_i^m}{\det(B)} \text{ for } i = 1 \ldots m \tag{4.28}$$

where $\boldsymbol{b}^m$ and A^m are the $(n+1)$-index column (i.e., the right-hand side) and submatrix (i.e., the left-hand side) of the mth-iteration augmented IPGE iterative matrix $[A|\boldsymbol{b}]^m$. Since $\boldsymbol{x}_B = \text{Adj}(B)\boldsymbol{b}/\det(B)$ from Cramer's Rule, where $\text{Adj}(B)$ is the adjunct matrix of B, we have that $\det(B)\boldsymbol{x}_B = \text{Adj}(B)\boldsymbol{b}$. This fact motivates the choice for solving $A\boldsymbol{x}' = \boldsymbol{b}'$ via forward and backward substitution rather than $A\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{x}' = \det(B)\boldsymbol{x}$ and $\boldsymbol{b}' = \det(B)\boldsymbol{b}$ (notice that $\det(B)$ is available from the REF factorization of A since $\det(B) = u_{m,m} = l_{m,m}$). Specifically, the substitutions applied to the scaled SLE provide an efficient mechanism to obtain $\boldsymbol{b}^m$, an integral vector, which together with $\det(B)$ provides the REF solution to the original SLE (see Equation (4.28)). Hence, a preliminary step of REF backward substitution is to

obtain $\boldsymbol{y}'$ as follows:

$$\boldsymbol{y}' = \det(B)\boldsymbol{y} = u_{m,m}\boldsymbol{y}$$

Then, the backward substitution for basic variables $\boldsymbol{x}'_B$ in $U\boldsymbol{x}' = \boldsymbol{y}'$ is given by:

$$x'_i = \frac{1}{u_{i,i}}\left(y'_i - \sum_{j=i+1}^{m} u_{i,j}x'_j\right) \text{ for } i = m..1. \tag{4.29}$$

Note that the sum goes only to index m because x_{m+1} to x_n equal 0 (i.e., they are nonbasic variables). Theorem 4.3.2 will demonstrate this backward substitution algorithm is REF. Then, Theorem 4.3.3 will demonstrate its bit-length upper bound is $\lceil 2m\log\sigma + (m+1)\log m\rceil = O(m\log(\sigma\sqrt{m}))$, that is, asymptotically equal to the IPGE bit-length bound.

Theorem 4.3.2. *The backward substitution for the first m elements of $(n\times 1)$-vector $\boldsymbol{x}'$ (i.e., $\boldsymbol{x}'_B$) in the equation $U\boldsymbol{x}' = \boldsymbol{y}'$, as specified by Equation* (4.29)*, is REF.*

Proof. We prove the correctness and REF property of Equation (4.29) by induction on k. For this purpose, let $r \in \mathbb{N}$ such that $r \leq m$.

Base case: $k = 2$. By default, we have $i = 1$ and from the IPGE formula:

$$a^2_{1,j} = \frac{\rho^2 a^1_{1,j} - a^1_{2,j}a^1_{1,2}}{\rho^1} = \frac{\rho^2 a^0_{1,j} - a^2_{2,j}a^0_{1,2}}{\rho^1}$$

where the second equality follows from the fact that $a^r_{r,j} = a^{r-1}_{r,j}$ for all r. Since this is an IPGE operation, it is free of roundoff errors.

Inductive step: $k = r > i$. Assume the statement holds for $k = 2\ldots r-1$. The proof of the inductive step is divided into two cases: (1) $i = r-1$, or (2) $i > r-1$.

Case (1): $i = r-1$. The result follows the same reasoning as the base case, since we

have:

$$a^r_{r-1,j} = \frac{\rho^r a^{r-1}_{r-1,j} - a^{r-1}_{r,j} a^{r-1}_{r-1,r}}{\rho^{r-1}} = \frac{\rho^r a^{r-2}_{r-1,j} - a^r_{r,j} a^{r-2}_{r-1,r}}{\rho^{r-1}}.$$

Case (2): $i < r-1$. Once again, according to the definition of IPGE:

$$a^r_{i,j} = \frac{\rho^r a^{r-1}_{i,j} - a^{r-1}_{r,j} a^{r-1}_{i,r}}{\rho^{r-1}} \tag{4.30}$$

$$= \frac{\rho^r \left(\rho^{r-1} a^{i-1}_{i,j} - \sum_{h=i+1}^{r-1} a^{r-1}_{h,j} a^{i-1}_{i,h} \right) - a^{r-1}_{r,j} \left(\rho^{r-1} a^{i-1}_{i,r} - \sum_{h=i+1}^{r-1} a^{r-1}_{h,r} a^{i-1}_{i,h} \right)}{\rho^{r-1}\rho^i} \tag{4.31}$$

$$= \frac{1}{\rho^{r-1}\rho^i} \left[\rho^{r-1} \left(\rho^r a^{i-1}_{i,j} - a^{r-1}_{r,j} a^{i-1}_{i,r} \right) - \sum_{h=i+1}^{r-1} \left(\rho^r a^{r-1}_{h,j} - a^{r-1}_{r,j} a^{r-1}_{h,r} \right) a^{i-1}_{i,h} \right] \tag{4.32}$$

$$= \frac{1}{\rho^{r-1}\rho^i} \left[\rho^{r-1} \left(\rho^r a^{i-1}_{i,j} - a^{r-1}_{r,j} a^{i-1}_{i,r} \right) - \sum_{h=i+1}^{r-1} \rho^{r-1} a^r_{h,j} a^{i-1}_{i,h} \right] \tag{4.33}$$

$$= \frac{1}{\rho^i} \left(\rho^r a^{i-1}_{i,j} - a^r_{r,j} a^{i-1}_{i,r} - \sum_{h=i+1}^{r-1} a^r_{h,j} a^{i-1}_{i,h} \right) = \frac{1}{\rho^i} \left(\rho^r a^{i-1}_{i,j} - \sum_{h=i+1}^{r} a^r_{h,j} a^{i-1}_{i,h} \right) \tag{4.34}$$

where the Equation (4.31) applied the inductive hypothesis on elements $a^{r-1}_{i,j}$ and $a^{r-1}_{i,r}$; where Equation (4.32) combined the first and second terms inside each parenthesis respectively; where Equation (4.33) made use of the IPGE definition of the product $\rho^{r-1} a^r_{h,j}$; and where in Expression (4.34), the first equation canceled the factor ρ^{r-1}, and the second equation included the second numerator term of the first equation into the sum of the third numerator term. This proves the correctness of the alternative formula for calculating $a^r_{i,j}$. It remains only to show that the formula is REF.

Equation (4.30) is REF because it is an IPGE operation. Since Equation (4.31) makes two substitutions, Equation (4.32) reorders some terms, and Equation (4.33) makes another substitution, this means the formula is still free of roundoff errors at this point. Moreover, obtaining the first equation of Expression (4.34) involved an exact division and its second equation simply reorganizing numerator terms. Thus,

the formula given by the second equation of Expression (4.34) is REF, and the theorem holds for $k = r$. Since $r \leq m$ was chosen arbitrarily, the result holds true for all k. □

Theorem 4.3.3. *The REF backward substitution algorithm requires a maximum bit-length that is asymptotically-equivalent to the IPGE entry maximum bit-length,* $\omega_{\max}$.

Proof. The elements of $\boldsymbol{y}$ and U have bit-lengths bounded by $\omega_{\max}$ since $y_i = a_{i,n+1}^{i-1}$ and $u_{i,j} = a_{i,j}^{i-1}$, for $1 \leq i \leq m$ and $1 \leq j \leq n$. Furthermore, from Equation (4.28), $x_i' = \det(B)x_i = b_i^m = a_{i,n+1}^m$ for $1 \leq i \leq m$. This implies the evaluation of the numerator in Equation (4.29) involves summing at most m products of two integers with bit-length $\omega_{\max}$ (including the computation of $\boldsymbol{y}' = \det(B)\boldsymbol{y}$). Therefore, the maximum magnitude of this numerator is given by:

$$m(\sigma^m m^{\frac{m}{2}})^2 = \sigma^{2m} m^{m+1}$$

thereby yielding a bit-length upper bound for REF backward substitution of $\lceil 2m \log \sigma + (m+1) \log m \rceil$, which alike $\omega_{\max}$ has $O(m \log(\sigma\sqrt{m}))$ length. □

It is now clear how the REF forward and backward substitution algorithms are utilized to solve $A\boldsymbol{x} = \boldsymbol{b}$ without roundoff error. Specifically, the substitutions are applied to the scaled system $A\boldsymbol{x}' = \boldsymbol{b}'$ in order to then obtain the exact solution quotient $\boldsymbol{x} = \boldsymbol{x}'/\det(B)$. Then, by storing the separate REF numerators of each variable and their REF common denominator, the solution may be calculated up to any level of precision.

4.3.3 Invalidity of Previous REF Substitution Algorithms

The forward substitution formula for $\boldsymbol{y}$ in $LD^{-1}\boldsymbol{y} = \boldsymbol{b}$ associated with the fraction-free LU factorization of full-row-rank matrix $A \in \mathbb{Z}^{n\times n}$ and right-hand vector $\boldsymbol{b} \in \mathbb{Z}^n$ presented in [90] is given by the following equation:

$$y_i = \frac{d_{i,i}}{l_{i,i}} \left(b_i - \sum_{j=1}^{i-1} l_{i,j} y_j \right) \tag{4.35}$$

where L and D^{-1} are two of the three matrix factors of the fraction-free factorization, $l_{i,j}$ is the (i,j)-entry of L, and $1 \leq j \leq i \leq n$. This equation mirrors the standard forward substitution algorithm given by Equation (4.21), with one crucial exception: the jth summand in the expression for y_i is $l_{i,j}y_j$ in Equation (4.35) while it is the fraction $l_{i,j}y_j/l_{j-1,j-1}l_{j,j}$ in Equation (4.21). It is straightforward to verify that the product of the respective symbolic matrix factors L and D^{-1} of the fraction-free LU factorization by Zhou and Jeffrey [90] and of our REF LU factorization (REF-LU) are equal. Since there are no factors in the ensuing recursive relation of the standard forward substitution for $\boldsymbol{y}$ in $LD^{-1}\boldsymbol{y} = \boldsymbol{b}$ that lead to the outright cancellation of $l_{j-1,j-1}l_{j,j}$ for all $1 \leq j < i \leq n$, the omission of said denominators in the forward substitution algorithm by Zhou and Jeffrey [90] is mathematically unjustifiable. This indicates that their fraction-free forward substitution algorithm is incorrect. Moreover, the associated proof is invalid because it follows from the false premise that the summands associated with the calculation of y_i, for all i, are integers.

The backward substitution algorithm presented in [90] is correct, but its correctness and fraction-free properties do not follow from the reasoning of its proof. In

particular, the key step of the proof gives the following sequence of equalities:

$$PA\boldsymbol{x} = LD^{-1}U\boldsymbol{x} = \det(A)LD^{-1}\boldsymbol{y} = \det(A)P\boldsymbol{b}$$

where P is a permutation matrix. From this expression, it is deduced that $\boldsymbol{x} = \text{Adj}(A)\boldsymbol{b}$, which is true based on Cramer's Rule. However, this only proves said backward substitution algorithm is algebraically correct (i.e., this proves that utilizing infinite precision or exact rational arithmetic the algorithm will give the exact solution), but it does not prove that the individual operations therein are fraction-free (i.e., preserve integrality), which is what Zhou and Jeffrey [90] claim to be proving. In fact, there is no mention of the specific operations of their backward substitution algorithm in the proof. Therefore, the correctness and fraction-free properties attributed to the backward substitution algorithm by Zhou and Jeffrey [90] do not follow from the reasoning in the associated proof.

4.4 Correspondence Between the REF and Traditional Factorizations

The REF substitution algorithms developed in Section 4.3 intimate why the functional forms of REF-LU and REF-Ch share the attributes of the traditional LU and Cholesky factorizations, respectively. This subsection will discuss these correspondences more formally, focusing primarily on explaining the Cholesky attributes of REF-Ch since those of the LU factorization can be similarly inferred.

4.4.1 Storage

The traditional Cholesky factorization of SPD $A \in \mathbb{Z}^{n\times n}$ requires storing $(n^2+n)/2$ individual entries, which represents half the number of entries needed by the traditional LU factorization of A. This subsection explains how the functional form of REF-Ch matches these storage requirements.

As mentioned by the previous subsections, the REF substitution algorithms also apply to REF-Ch because the U matrix from REF-LU equals the transpose of the L matrix for a SPD input matrix. Taking this relationship into account, The REF substitution algorithms given by Expression (4.27) and Equation (4.29) demonstrate that only the lower-triangular section of L needs to be stored in order to apply REF forward and backward substitution (or to apply traditional forward and backward substitution, for that matter). This is because instead of storing D explicitly, the substitution algorithms generate its entries on the fly from the diagonal elements of L. Since L has $(n^2+n)/2$ lower-triangular entries, therefore, the number of stored elements of the functional form of REF-Ch match those of its traditional counterpart. Similarly, the functional form of REF-LU of A requires storing only the n^2+n total entries of L and U, as in the traditional LU factorization. Notice that the functional form of REF-LU requires twice the storage of REF-Ch and, therefore, the latter factorization shares the storage advantage of the traditional Cholesky factorization.

4.4.2 Number of Operations

The construction of the traditional Cholesky factorization requires $n^3/3$ (floating-point) operations, which represents half the operations needed by the traditional LU factorization [40]. This subsection explains how the functional form of REF-Ch replicates these requirements up to a constant multiple.

Based on the contents of L, the number of operations required to construct the functional form of REF-Ch of SPD $A \in \mathbb{Z}^{n\times n}$, is calculated by counting the number of operations of applying a reduced version of IPGE to A. In particular, since $l_{i,j} = a_{i,j}^{j-1}$ for $j \leq i$ and $l_{i,j} = 0$ otherwise, where $a_{i,j}^{j-1}$ denotes the (i,j)-entry of the $(j$–$1)$-iteration matrix of IPGE, it is only necessary to obtain entries $a_{i>k,j>k}^{k}$ during iteration k of IPGE, where $0 < k < n$, and $i, j \leq n$ (i.e., IPGE pivoting

operations are performed only on the elements below and to the right of the pivot element). Consequently, accounting for the fact that the calculation of each IPGE entry comprises four operations, the required number of operations to calculate L is given by:

$$4\sum_{h=1}^{n-1} h^2 = \frac{4n^3 - 6n^2 + 2n}{3} \leq \frac{4n^3}{3}.$$

This is analogous to the $n^3/3$ operations required to construct a traditional Cholesky factorization. Similarly, at most $8n^3/3$ operations are needed to calculate L and U of the functional form of REF-LU of A, which equals twice the number of operations needed by the functional form of REF-Ch. Hence, the functional version of REF-Ch matches the advantage of the traditional Cholesky factorization in terms of number of operations as well.

A similar analysis of the REF substitution algorithms given by Expression (4.27) and Equation (4.29), shows that the number of operations to perform REF forward and backward substitution replicates that of traditional forward and backward substitution up to a constant multiple. The storage and number of operations associated with the functional forms of the REF factorizations, thus, explain the connection of the REF-Ch and REF-LUs with their traditional counterparts.

4.5 Computational Complexity

Standard data structures (e.g., doubles, fixed-precision integers, etc.) have constant bit-length and, hence, they do not factor into the computational complexity of algorithms that use them. Thus, the computational costs associated with the calculation and implementation of traditional LU and Cholesky factorizations on a square matrix of order n equal their number of operations, which are $O(n^3)$ and $O(n^2)$, respectively. Associated with these simplified costs, however, is the potential for inaccurate algorithm output due to the impact of endemic roundoff errors, as

mentioned in the introductory part of this section (and at greater length in Section 1). Conversely, the REF LU and REF Cholesky factorizations and the related REF substitution algorithms guarantee exactness. In exchange for accuracy, as is the case for any other REF algorithm, the REF factorizations and substitutions require working with operands whose individual bit-length is not fixed (as Sections 4.2 and 4.3 demonstrate, however, the bit-length of the REF factorizations and their accompanying REF substitution algorithms is bounded polynomially). Hence, the total computational costs of calculating and using the REF factorizations must include both their number of operations and the added complexity of the operands' growth. In order to provide the complexity of the REF factorizations, we will also derive the complexity results for IPGE. For the ensuing complexity discussion, recall that σ is the entry with the maximum absolute value in the input matrix.

Lee and Saunders [53] calculated $O(n^5 \log^2 \sigma)$ as the total cost of performing IPGE on fully-dense $n \times n$ matrices with individual entries taken from the polynomial domain. Their calculations assume the bit-length of each entry of IPGE during iteration k equals $k \log \sigma$, in concordance with the dense univariate model of polynomial computation of Gentleman and Johnson [35]. However, because this computational model ignores coefficient growth, it is our belief—and that of Dr. B. David Saunders, as verified via a private communication—that when considering the integral domain, it is more appropriate to utilize the bit-length determined by Hadamard's inequality (see Section 2.3.5.2) to derive the worst-case complexity of IPGE. Consequently, we update IPGE's computational complexity to reflect this more suitable maximum bit-length bound.

To be precise, the complexity measures we derive apply to matrices whose entries are drawn from the integral domain. Moreover, our derivations take $O(w \log w \log \log w)$ as the cost of multiplying/dividing two integers of bit-length w according to the best

known Fast Fourier transform algorithms [50, 71].

Utilizing the worst-case bit-length of each IPGE entry—denoted as $\mathbf{H}(\omega_{\max})$ based on its connection with Hadamard's inequality—the worst-case complexity (WCC) of IPGE, which matches that of the REF factorizations (see Section 4.4.2), is calculated as:

$$\begin{aligned}\text{WCC(IPGE/REF-LU/REF-Ch)} &= O(n^3[\mathbf{H}(\omega_{\max}) \log \mathbf{H}(\omega_{\max}) \log\log \mathbf{H}(\omega_{\max})]) \\ &= O\left(n^4 \max(\log^2 n \log\log n, \log^2 \sigma \log\log \sigma)\right)\end{aligned}$$

where the first equality accounts for the $O(n^3)$ multiplication/division operations (which dominate the costs of addition/substraction), whose individual operands have bit-lengths of at most $\mathbf{H}(\omega_{\max}) = \lceil n \log(\sigma\sqrt{n}) \rceil$. Similarly, since the bit-lengths of the REF substitution algorithms are asymptotically equal to those of IPGE (see Section 4.3), the computational complexity of REF forward and backward substitution is given by:

$$\begin{aligned}\text{WCC(REF Substitution)} &= O(n^2[\mathbf{H}(\omega_{\max}) \log \mathbf{H}(\omega_{\max}) \log\log \mathbf{H}(\omega_{\max})]) \\ &= O\left(n^3 \max(\log^2 n \log\log n, \log^2 \sigma \log\log \sigma)\right)\end{aligned}$$

where the first equality accounts for the $O(n^2)$ multiplication/division operations and the IPGE worst-case bit-length of the operands. As Section 2.3.5.2 explains, however, Hadamard's inequality yields a pessimistic bit-length for IPGE. Hence, in practice, the typical costs of IPGE and our REF algorithms could be noticeably lower than the above costs.

We note that for special types of sparse matrices, the costs of performing IPGE can be reduced by a factor of up to $O(n^2)$ utilizing the more efficient form of IPGE

developed by Lee and Saunders [53]. Hence, when calculating the REF factorizations on certain sparse matrices, the above computational costs can be reduced by a factor of up to $O(n^2)$ as well.

4.6 Conclusions

We remark that there exist methods for solving integer and rational linear systems exactly other than the REF LU factorization approach or REF Gaussian elimination approaches, in general. The most prominent of these alternatives are described in Section 2.3 and they are divided roughly into three categories: p-adic methods—e.g., [21], [62], [23]—, black-box linear algebra methods—e.g., [86], [47], [46]—, and iterative numerical methods—e.g., [84], [38]. Briefly stated, the first two classifications of these major alternative approaches prioritize space complexity, while the third seeks to lower the number of operations performed. We refer the reader to Cook and Steffy [17], which features a comparison of the solution run-times attained via an algorithm from each of the four main categories in the context of linear programming (i.e., including an exact rational-arithmetic LU factorization-based method); Section 5.1 briefly reports some of its findings. The computational performance of each algorithm tested therein is somewhat dependent on the specific instance being solved, although the iterative numerical and black-box linear algebra methods performed the best and the worst, respectively, in general. We believe a similar comparison that implements the REF algorithms herein presented appropriately would be a worthwhile avenue for future research.

That said, the motivation for developing and continuing to enhance the REF factorizations and the associated REF substitution algorithms is based on their potential to adapt to current simplex-based LP solvers and mixed-integer programming solvers. Indeed, a major goal of our ongoing research is to craft algorithms that can

be ultimately integrated into existing LP solvers in order to equip them with efficient tools for avoiding some of the inconsistent outputs discussed in Section 1. A prospective implementation of the algorithms herein presented would follow a blueprint similar to the development of the exact LP solver `QSopt_ex` [5]. This solver enhanced the open-source `QSopt` LP solver [3] by adding an improved version of the exact-arithmetic validation algorithm pioneered by Dhiflaoui et al. [20]. In particular, as Section 3.3.3 explains, `QSopt_ex` computes and applies an exact rational-arithmetic LU factorization to verify the validity of the floating-point simplex solution to a given rational LP; the solver iteratively increases the floating-point precision and restarts the simplex algorithm whenever the solver-provided solution is invalid. Hence, future research will look into enhancing REF-LU and modifying the validation subroutines of `QSopt_ex` accordingly.

5. COMPARISONS OF TECHNIQUES FOR BASIC SOLUTION VALIDATION

5.1 Recent Enhancements and Computational Tests in the Literature

In order to provide an up-to-date perspective for the ensuing discussion and computational experiments, this section begins by describing recent key enhancements to the p-adic lifting and iterative refinement approaches for solving SLEs exactly (described in Sections 2.3.2 and 2.3.3, respectively). In particular, in his implementation of these two approaches, Steffy [74] replaces the respective standard rational reconstruction subroutines, which are required for recovering the exact solution from the algorithms' final outputs, with the output sensitive lifting strategy (see Section 2.3.4). This modification increases the speed and efficiency of both methods mainly by integrating rational reconstruction in their intermediate steps and by allowing smaller solution size bounds to be employed in the reconstruction process. The author also adds the use of the DLCM technique (i.e., solution-denominator lowest common multiple), which accelerates component-wise reconstruction by taking advantage of the fact that the solution vector's denominators share common factors; this technique is also used by the numerical libraries LinBox[1] [22] and NTL[2] [73].

Through the above-mentioned enhancements, the computational efficiency of the p-adic lifting and iterative refinement approaches can be significantly boosted. Indeed, the computational tests performed by Steffy [74] demonstrate that the performances of both approaches are improved by several orders of magnitude in some cases. In said study the instances are random nonsingular matrices generated from different dense matrix templates. As mentioned therein, however, these competitive results are associated with problems whose final solution size is small, which is not

[1]Project LinBox — Exact Computational Linear Algebra. http://www.linalg.org.
[2]NTL — Number Theory Library. http://www.shoup.net/ntl.

a property that can be determined a priori. Moreover, the two underlying assumptions for applying iterative refinement for solving SLEs exactly via this design are that the final solution can be sufficiently approximated in floating-point and that the inversion or factorization of the matrix is not too numerically ill-conditioned.

Cook and Steffy [17] evaluated, within the context of linear programming, four of the prominent approaches outlined in Section 2 for solving SLEs exactly: rational arithmetic LU factorization, Wiedemann's black box algorithm, p-adic lifting, and iterative refinement. Their implementation of the latter two approaches followed largely the same blueprint as [74]—described in the preceding paragraphs—with some further improvements. One additional enhancement is their use of Lehmer's gcd algorithm [54], which performs divisions on floating-point approximations of large integers (rather than on the integers themselves) in the execution of the extended Euclidian algorithm; it is used in similar fashion in [16]. Another one of their enhancements is the reduction of calls to rational reconstruction from every 10 algorithm loops, as executed by Chen and Storjohann [15], to every power-of-2 loops; this modification was found to speed computation in preliminary experiments.

The four exact SLE solution approaches were tested in [17] using 276 sparse instances from various popular linear programming libraries. The assembled problem testbed consisted of each instance's optimal or best known basis yielded within 24 hours by the exact solver `QSopt_ex` as well as a corresponding RHS vector. The experiments record the individual solve run-times of the four approaches normalized by the run-time of the p-adic lifting approach, and then they compute the geometric means over all instances and over cohesive groups of instances. In all, iterative refinement had the best average performance, although the exhibited results exclude the five instances where it failed due to numerical issues—the three other approaches were able to solve every instance to completion. The p-adic lifting approach followed

closely behind it with run-times that were approximately 15% slower. Despite its use of rational arithmetic, the exact LU factorization method was on average only three times slower than both of these methods. This performance is encouraging for the prospects of improved exact LU factorization methods since it is not far from striking distance of the top two approaches, which benefitted from the various enhancements listed in [15, 17, 74] and summarized in the preceding paragraphs. Hence, it is realistic to expect that the proposed REF factorization framework could significantly shrink and possibly overturn this performance gap in many cases. On the other hand, appreciably more work would be required for Wiedemman's black-box algorithm to become competitive in this context since its run-times were approximately 40 times slower than the top two algorithms.

Other key takeaways from the computational results in [17] reinforce the case for implementing the exact LU approach for solving SLEs in the context of basic solution validation for exact linear programming. For starters, as attested by its inability to solve five instances in said study, iterative refinement is not suitable as a general exact SLE solution subroutine. The authors highlight this point by stating that it is more appropriate to apply iterative refinement when the SLE is known to be numerically stable, which is of course not easily predictable when solving arbitrary LPs. Accordingly, they advocate the use of p-adic lifting since its performance is similar yet more robust than that of iterative refinement. They also promote the use of the exact rational arithmetic LU factorization approach since its code turns out to be faster than p-adic lifting for a critical subset of the tested instances, namely those with moderate to large solution size. This matches the previous observations in [15, 74] that the best performances of p-adic lifting (and iterative refinement) are most directly linked with simpler problems that have small solution sizes. This seems to indicate that the exact LU factorization approach is best equipped for efficiently

solving SLEs that are numerically difficult and/or intricate—i.e., having relatively larger solution sizes.

The most decisive argument for implementing the exact LU factorization approach for basic solution validation is voiced by the current practice of state-of-the-art exact LP solvers `QSopt_ex` 2.6[3] and `SoPlex` 2.2.0[4]. Both of these solvers implement the exact rational arithmetic LU factorization and not p-adic lifting for basic solution validation. To be precise, the exact LP `SoPlex` extension—which uses LP iterative refinement to obtain increasingly accurate approximate solutions as Section 3.3.4 explains—also provides output sensitive lifting as an option for obtaining exact solutions. However, in Gleixner [37], which contains the study and analysis attached to the referenced `SoPlex` release, the author found that the LU factorization method was 46% faster than output sensitive lifting over all the tested instances; conversely, output sensitive lifting was 5% faster than the LU factorization method only for instances that took less than .5 seconds to solve.

The preceding analysis strongly supports the view that the exact LU factorization approach should be the go-to tool for basic solution validation within exact LP. As has been explained thus far, there are other shortcut techniques that may be used in place of this approach, such as rational reconstruction and p-adic lifting, which achieve quicker run-times when they are applied to simpler instances. However, these techniques become more costly/ineffective than the factorization-based approach when the problem is numerically unstable and/or when the solution vector coefficients require relatively larger bit-lengths for their exact expression [17]. A problematic aspect of the latter statement is that solution size is both independent of the numerical condition of the basis matrix and difficult to determine a priori.

[3]Release notes available at http://www.math.uwaterloo.ca/~bico/qsopt/ex.
[4]Release notes available at http://soplex.zib.de/notes-220.txt.

Thus, it is possible that a seemingly ordinary RHS vector may cause these methods to achieve their worst-case performance for ill-conditioned and well-conditioned basis matrices alike. Since the overarching objective of this dissertation is to eliminate the potentially pernicious aftereffects of roundoff errors efficiently, the subsequent computational tests in this work consider the rational arithmetic LU factorization method as the primary point of reference for basic solution validation. Nevertheless, reiterating a statement from Section 3.4, the most competitive and effective exact LP implementations will be those that apply a combination of the best known techniques—a plausible implementation could attempt rational reconstruction as a first measure of exactness validation and, when this initial check is unsuccessful, exact basic solution validation. Having said that, the integration of the featured framework with these shortcut alternatives is left for future work.

5.2 Featured Computational Tests

As mentioned in previous sections, the central function of the REF factorization framework is the efficient validation of basic solutions for use in state-of-the-art exact LP solvers. An indispensable task in proving the competitiveness of the featured algorithms within this context is to assess their performance compared to those involved in the construction and triangular solves of the in-use exact rational arithmetic LU factorization approach. Hence, the rest of this section describes the various experiments performed to measure the computational performance of the REF factorization framework in relation to this principal method. Additionally, an efficient adaptation of the Q-matrix revised simplex method is devised and tested.

The metrics of interest for this study are threefold: (1) run-time of the construction process of each exact tool; (2) run-time of the respective exact solution processes, and (3) storage requirements. Original derivations and useful theoretical

analyses accompany each set of computational tests.

5.2.1 Computing Environment

The experiments herein described were performed on a machine equipped with 32GB of RAM memory and a quad-core 3.6GHz Intel E5-1620 processor. The operating system of the machine is CentOS Linux 7 (1511), which is derived from Red Hat Enterprise Linux 7.2. The code was written in C++ using the unlimited-precision GNU-GMP numerical library [42].

5.2.2 General Instance Specifications

The following instance specifications are standard to all the featured computational tests, unless noted otherwise. The input LHS matrices of the linear systems are randomly generated fully-dense square matrices. In the exact rational arithmetic LU factorization comparisons, the dimension of the LHS matrices is $n \in [50, 500]$; the upper limit was set to 500 due to the high run-times required by the rational arithmetic factorizations according to preliminary runs. In the Q-matrix method comparisons, the dimension of the LHS matrices is $n \in [100, 1000]$. The input RHS matrices are randomly generated and have dimension $n \times 50$, meaning the triangular solve tests are applied to 50 distinct RHS vectors at a time per repetition; this choice was made to obtain stable results since, for small n, the recorded run-times for solving few RHS vectors can be insignificant or inconsistent. The input matrix entries are integers drawn uniformly from the interval $[-99, 99]$ (excluding zero). For each setting of the dimension parameter n, the experiments perform 30 repetitions and report the arithmetic mean and standard deviation of each metric of interest.

Before proceeding with the individual experiments and results, it is worthwhile to elaborate on the choice of input instances. As previous sections state, the overarching objective of this plan of study is to eliminate the rational arithmetic operations

required in the standard exact LU factorization approach by replacing them with integer arithmetic operations via the REF factorization framework. To determine the computational savings due directly to this fundamental change, it is necessary to ignore the run-time spent searching for a nonzero pivot element and performing the corresponding row and column permutations. The most effective way to exclude them is to avoid pivot searches altogether and, hence, our experiments work entirely with fully-dense matrices (in the rare events in which a zero was introduced by the algorithms and then selected as a pivot, the corresponding run was discarded). However, since numerical stability is not an issue when working in unlimited precision, the exact rational arithmetic LU factorizations can and would likely draw their pivots from the same matrix coordinates as the REF LU factorization when working with sparse instances. The ensuing subsection elucidates the latter point by examining the rational arithmetic LU factorizations in greater detail and by establishing their connection to the REF LU factorization. The task of optimizing the REF framework algorithms for testing them on sparse instances is left for future work.

5.3 Comparison with the Exact Rational Arithmetic LU Factorization Approach

5.3.1 Analysis of the Doolittle and Crout Factorizations

As Section 2.2.3 explains, the Doolittle and Crout Factorizations are two types of unique LU factorizations obtained by requiring L to be unit lower-triangular and U to be unit upper-triangular, respectively. The distinct elementary algorithms used to construct them can be found in [33] and [66]. It is important to realize that the two factorizations—as well as any other exact rational arithmetic LU factorization—would solve the linear system $A\boldsymbol{x} = \boldsymbol{b}$ without accruing roundoff errors using the same procedure. Expressly, as Section 2.2.3 explains, each performs a triangular solve of the (lower-triangular) SLE $L\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y} \in \mathbb{Q}^n$ followed by a triangular

solve of the (upper-triangular) SLE $U\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x} \in \mathbb{Q}^n$; both triangular solves must be done in exact rational arithmetic to avoid roundoff errors. Moreover, based on Cramer's rule, the outcome of this process can be expressed exactly as:

$$\boldsymbol{x} = \frac{\mathrm{Adj}(A)\boldsymbol{b}}{\det(A)}$$

where $\mathrm{Adj}(A)$ and $\det(A)$ are the adjunct matrix and determinant of nonsingular matrix A, respectively. A key related point is that each resulting exact rational $x_i = [\mathrm{Adj}(A)_{(i,:)}\boldsymbol{b}]/\det(A)$, for $1 \leq i \leq n$, can be simplified when its numerator and denominator share common factors—i.e., in such a situation the requisite bit-length of x_i is smaller than the sum of the bit-lengths of $\mathrm{Adj}(A)_{(i,:)}\boldsymbol{b}$ and $\det(A)$. Then again, it is plausible that at least one of the solution numerators is relatively prime to $\det(A)$. This is noteworthy because, generally speaking, the bit-length of such a rational would be among the largest required by an individual entry of the solution vector. To inspect this issue further, an experiment described Section 5.3.2.2 counts the number of instances in which at least one element of $\boldsymbol{x}$ had to store $\det(A)$ as its denominator because the exact rational solution entry could not be simplified.

Having clarified the exact rational arithmetic solution process, this subsection establishes the connection between the REF LU factorization and the exact forms of the two unique rational arithmetic LU factorizations, hereafter alternatively referred to as Doolittle-LU and Crout-LU. This connection allows the entries of Doolittle-LU and Crout-LU to be expressed in terms of IPGE entries and in terms of subdeterminants of A. To this end, it will be useful to introduce the following notation.

Definition 9. *Let α_C^R be an abbreviation of $\det(A)_C^R$, the subdeterminant involving the rows and columns of A indexed by sets $R, C \subseteq \{1 \ldots n\}$ such that $|R| = |C|$.*

For the ensuing discussion, let $a_{i,j}^k$ again denote the kth-iteration (i,j)-entry asso-

ciated with applying IPGE to $A \in \mathbb{Z}^{n\times n}$ according to Assumption 2—which specifies that the kth IPGE pivot, ρ^k, is always given by $a_{k,k}^{k-1} \neq 0$—for $1 \leq i, j \leq n$ and $0 \leq k \leq n$. In addition, fix L, D^{-1}, and U to be the three matrices comprising the REF LU factorization of A.

Theorem 5.3.1. *The Doolittle Factorization can be readily obtained from the REF LU factorization given by Equation* (4.11)*.*

Proof. Expanding the REF-LU diagonal matrix gives that:

$$LD^{-1}U = L \begin{bmatrix} \frac{1}{a_{1,1}^0} & & & & 0 \\ & \frac{1}{a_{1,1}^0 a_{2,2}^1} & & & \\ & & \frac{1}{a_{2,2}^1 a_{3,3}^2} & & \\ & & & \ddots & \\ 0 & & & & \frac{1}{a_{n-1,n-1}^{n-2} a_{n,n}^{n-1}} \end{bmatrix} U \tag{5.1}$$

$$= L \begin{bmatrix} \frac{1}{a_{1,1}^0} & & & & 0 \\ & \frac{1}{a_{2,2}^1} & & & \\ & & \frac{1}{a_{3,3}^2} & & \\ & & & \ddots & \\ 0 & & & & \frac{1}{a_{n,n}^{n-1}} \end{bmatrix} \begin{bmatrix} 1 & & & & 0 \\ & \frac{1}{a_{1,1}^0} & & & \\ & & \frac{1}{a_{2,2}^1} & & \\ & & & \ddots & \\ 0 & & & & \frac{1}{a_{n-1,n-1}^{n-2}} \end{bmatrix} U \tag{5.2}$$

$$= \begin{bmatrix} 1 & & & & 0 \\ \frac{a_{2,1}^0}{a_{1,1}^0} & 1 & & & \\ \frac{a_{3,1}^0}{a_{1,1}^0} & \frac{a_{3,2}^1}{a_{2,2}^1} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ \frac{a_{n,1}^0}{a_{1,1}^0} & \frac{a_{n,2}^1}{a_{2,2}^1} & \frac{a_{n,3}^2}{a_{3,3}^2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_{1,1}^0 & a_{1,2}^0 & a_{1,3}^0 & \cdots & a_{1,n}^0 \\ & \frac{a_{2,2}^1}{a_{1,1}^0} & \frac{a_{2,3}^1}{a_{1,1}^0} & \cdots & \frac{a_{2,n}^1}{a_{1,1}^0} \\ & & \frac{a_{3,3}^2}{a_{2,2}^1} & \cdots & \frac{a_{3,n}^2}{a_{2,2}^1} \\ & & & \ddots & \vdots \\ 0 & & & & \frac{a_{n,n}^{n-1}}{a_{n-1,n-1}^{n-2}} \end{bmatrix}; \tag{5.3}$$

where Equation (5.2) splits D^{-1} into two diagonal matrices by factoring out the lower-

indexed IPGE entries and placing them into a new left-hand matrix; and, where Equation (5.3) right-multiplies L by the new left-hand diagonal matrix and left-multiplies U by the modified right-hand diagonal matrix. The result follows from the fact that there exists only one LU factorization with a unit lower-triangular matrix and, consequently, Equation (5.3) is equivalent to the Doolittle LU factorization. □

Corollary 3. *The individual entries of Doolittle-LU are expressed as subdeterminants of A as follows:*

$$LU = \begin{bmatrix} 1 & & & & & 0 \\ \frac{\alpha_1^2}{\alpha_1^1} & 1 & & & & \\ \frac{\alpha_1^3}{\alpha_1^1} & \frac{\alpha_{1,2}^{1,3}}{\alpha_{1,2}^{1,2}} & 1 & & & \\ \frac{\alpha_1^4}{\alpha_1^1} & \frac{\alpha_{1,2}^{1,4}}{\alpha_{1,2}^{1,2}} & \frac{\alpha_{1,2,3}^{1,2,4}}{\alpha_{1,2,3}^{1,2,3}} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \ddots & \\ \frac{\alpha_1^n}{\alpha_1^1} & \frac{\alpha_{1,2}^{1,n}}{\alpha_{1,2}^{1,2}} & \frac{\alpha_{1,2,3}^{1,2,n}}{\alpha_{1,2,3}^{1,2,3}} & \frac{\alpha_{1,2,3,4}^{1,2,3,n}}{\alpha_{1,2,3,4}^{1,2,3,4}} & \cdots & 1 \end{bmatrix} \begin{bmatrix} \alpha_1^1 & \alpha_2^1 & \alpha_3^1 & \alpha_4^1 & \cdots & \alpha_n^1 \\ & \frac{\alpha_{1,2}^{1,2}}{\alpha_1^1} & \frac{\alpha_{1,3}^{1,2}}{\alpha_1^1} & \frac{\alpha_{1,4}^{1,2}}{\alpha_1^1} & \cdots & \frac{\alpha_{1,n}^{1,2}}{\alpha_1^1} \\ & & \frac{\alpha_{1,2,3}^{1,2,3}}{\alpha_{1,2}^{1,2}} & \frac{\alpha_{1,2,4}^{1,2,3}}{\alpha_{1,2}^{1,2}} & \cdots & \frac{\alpha_{1,2,n}^{1,2,3}}{\alpha_{1,2}^{1,2}} \\ & & & \frac{\alpha_{1,2,3,4}^{1,2,3,4}}{\alpha_{1,2,3}^{1,2,3}} & \cdots & \frac{\alpha_{1,2,3,n}^{1,2,3,4}}{\alpha_{1,2,3}^{1,2,3}} \\ & & & & \ddots & \vdots \\ 0 & & & & & \frac{\alpha_{1,\ldots,n}^{1,\ldots,n}}{\alpha_{1,\ldots,n-1}^{1,\ldots,n-1}} \end{bmatrix}. \quad (5.4)$$

To highlight the similarity between the Doolittle factorization and the REF LU factorization, we also express the contents of the L and U matrices of REF-LU as subdeterminants of A as follows:

$$LU = \begin{bmatrix} \alpha_1^1 & & & & & 0 \\ \alpha_1^2 & \alpha_{1,2}^{1,2} & & & & \\ \alpha_1^3 & \alpha_{1,2}^{1,3} & \alpha_{1,2,3}^{1,2,3} & & & \\ \alpha_1^4 & \alpha_{1,2}^{1,4} & \alpha_{1,2,3}^{1,2,4} & \alpha_{1,2,3,4}^{1,2,3,4} & & \\ \vdots & \vdots & \vdots & \ddots & \ddots & \\ \alpha_1^n & \alpha_{1,2}^{1,n} & \alpha_{1,2,3}^{1,2,n} & \alpha_{1,2,3,4}^{1,2,3,n} & \cdots & \alpha_{1,\ldots,n}^{1,\ldots,n} \end{bmatrix} \begin{bmatrix} \alpha_1^1 & \alpha_2^1 & \alpha_3^1 & \alpha_4^1 & \cdots & \alpha_n^1 \\ & \alpha_{1,2}^{1,2} & \alpha_{1,3}^{1,2} & \alpha_{1,4}^{1,2} & \cdots & \alpha_{1,n}^{1,2} \\ & & \alpha_{1,2,3}^{1,2,3} & \alpha_{1,2,4}^{1,2,3} & \cdots & \alpha_{1,2,n}^{1,2,3} \\ & & & \alpha_{1,2,3,4}^{1,2,3,4} & \cdots & \alpha_{1,2,3,n}^{1,2,3,4} \\ & & & & \ddots & \vdots \\ 0 & & & & & \alpha_{1,\ldots,n}^{1,\ldots,n} \end{bmatrix}. \quad (5.5)$$

Notice that, except for the diagonal elements of each respective lower-triangular

matrix, the entries of the REF L and U matrices of Equation (5.5) are exactly the numerators of the corresponding entries of the factorization given by Equation (5.4). This observation leads to the prediction that, in general, Doolittle-LU requires roughly twice the storage of REF-LU. The ensuing theorem and corollary help lead to the same conjecture with respect to the relationship between Crout-LU and REF-LU.

Theorem 5.3.2. *The Crout Factorization can be readily obtained from the REF LU factorization given by Equation* (4.11).

Proof. Expanding the REF-LU diagonal matrix gives that:

$$LD^{-1}U = L \begin{bmatrix} \frac{1}{a^0_{1,1}} & & & & 0 \\ & \frac{1}{a^0_{1,1}a^1_{2,2}} & & & \\ & & \frac{1}{a^1_{2,2}a^2_{3,3}} & & \\ & & & \ddots & \\ 0 & & & & \frac{1}{a^{n-2}_{n-1,n-1}a^{n-1}_{n,n}} \end{bmatrix} U \tag{5.6}$$

$$= L \begin{bmatrix} 1 & & & & 0 \\ & \frac{1}{a^0_{1,1}} & & & \\ & & \frac{1}{a^1_{2,2}} & & \\ & & & \ddots & \\ 0 & & & & \frac{1}{a^{n-2}_{n-1,n-1}} \end{bmatrix} \begin{bmatrix} \frac{1}{a^0_{1,1}} & & & & 0 \\ & \frac{1}{a^1_{2,2}} & & & \\ & & \frac{1}{a^2_{3,3}} & & \\ & & & \ddots & \\ 0 & & & & \frac{1}{a^{n-1}_{n,n}} \end{bmatrix} U \tag{5.7}$$

$$= \begin{bmatrix} a^0_{1,1} & & & & 0 \\ a^0_{2,1} & \frac{a^1_{2,2}}{a^0_{1,1}} & & & \\ a^0_{3,1} & \frac{a^1_{3,2}}{a^0_{1,1}} & \frac{a^2_{3,3}}{a^1_{2,2}} & & \\ \vdots & \vdots & \vdots & \ddots & \\ a^0_{n,1} & \frac{a^1_{n,2}}{a^0_{1,1}} & \frac{a^2_{n,3}}{a^1_{2,2}} & \cdots & \frac{a^{n-1}_{n,n}}{a^{n-2}_{n-1,n-1}} \end{bmatrix} \begin{bmatrix} 1 & \frac{a^0_{1,2}}{a^0_{1,1}} & \frac{a^0_{1,3}}{a^0_{1,1}} & \cdots & \frac{a^0_{1,n}}{a^0_{1,1}} \\ & 1 & \frac{a^1_{2,3}}{a^1_{2,2}} & \cdots & \frac{a^1_{2,n}}{a^1_{2,2}} \\ & & 1 & \ddots & \frac{a^2_{3,n}}{a^2_{3,3}} \\ & & & \ddots & \vdots \\ 0 & & & & 1 \end{bmatrix} \tag{5.8}$$

where Equation (5.7) splits D^{-1} into two diagonal matrices by factoring out the lower-indexed IPGE entries and placing them into a new right-hand matrix; and, where Equation (5.8) left-multiplies U by the new right-hand diagonal matrix and right-multiplies L by the modified left-hand diagonal matrix. The result follows from the fact that there exists only one LU factorization with a unit upper-triangular matrix and, consequently, Equation (5.8) is equivalent to the Crout LU factorization. □

Corollary 4. *The individual entries of Crout-LU are expressed as subdeterminants of A as follows:*

$$LU = \begin{bmatrix} \alpha_1^1 & & & & & 0 \\ \alpha_1^2 & \frac{\alpha_{1,2}^{1,2}}{\alpha_1^1} & & & & \\ \alpha_1^3 & \frac{\alpha_{1,2}^{1,3}}{\alpha_1^1} & \frac{\alpha_{1,2,3}^{1,2,3}}{\alpha_{1,2}^{1,2}} & & & \\ \alpha_1^4 & \frac{\alpha_{1,2}^{1,4}}{\alpha_1^1} & \frac{\alpha_{1,2,3}^{1,2,4}}{\alpha_{1,2}^{1,2}} & \frac{\alpha_{1,2,3,4}^{1,2,3,4}}{\alpha_{1,2,3}^{1,2,3}} & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ \alpha_1^n & \frac{\alpha_{1,2}^{1,n}}{\alpha_1^1} & \frac{\alpha_{1,2,3}^{1,2,n}}{\alpha_{1,2}^{1,2}} & \frac{\alpha_{1,2,3,4}^{1,2,3,n}}{\alpha_{1,2,3}^{1,2,3}} & \cdots & \frac{\alpha_{1,\ldots,n}^{1,\ldots,n}}{\alpha_{1,\ldots,n-1}^{1,\ldots,n-1}} \end{bmatrix} \begin{bmatrix} 1 & \frac{\alpha_2^1}{\alpha_1^1} & \frac{\alpha_3^1}{\alpha_1^1} & \frac{\alpha_4^1}{\alpha_1^1} & \cdots & \frac{\alpha_n^1}{\alpha_1^1} \\ & 1 & \frac{\alpha_{1,3}^{1,2}}{\alpha_{1,2}^{1,2}} & \frac{\alpha_{1,4}^{1,2}}{\alpha_{1,2}^{1,2}} & \cdots & \frac{\alpha_{1,n}^{1,2}}{\alpha_{1,2}^{1,2}} \\ & & 1 & \frac{\alpha_{1,2,4}^{1,2,3}}{\alpha_{1,2,3}^{1,2,3}} & \cdots & \frac{\alpha_{1,2,n}^{1,2,3}}{\alpha_{1,2,3}^{1,2,3}} \\ & & & 1 & \cdots & \frac{\alpha_{1,2,3,n}^{1,2,3,4}}{\alpha_{1,2,3,4}^{1,2,3,4}} \\ & & & & \ddots & \vdots \\ 0 & & & & & 1 \end{bmatrix}.$$

5.3.2 Experiments and Results

The present subsection reports and analyzes the results of computational tests designed to compare the algorithms used to construct REF-LU, Doolittle-LU, and Crout-LU and to perform the respective triangular solves. In each corresponding algorithm, the appropriate standard integer or rational mathematical operations of the `GMP` library are employed. However, in the case of the REF factorization framework algorithms, we use the version of Jebelean's exact division algorithm [44] offered within the `GMP` library to take advantage of IPGE Property 1 and of the theoretical results derived in Section 4. Specifically, this special division algorithm, which halves

the work of a normal basecase division [42], can be employed by the REF factorization algorithms because their constituent divisions are guaranteed to be exact.

5.3.2.1 Run-Times

The first experiment consists of measuring the run-times required to construct the three LU factorizations and to perform the exact triangular solves. As a point of clarification, in this experiment Doolittle-LU and Crout-LU are explicitly constructed using their eponymous algorithms (see [33] and [66]) and not the derivations described by Theorems 5.3.1 and 5.3.2. Table 5.1 provides a summary of the run-times in seconds (s) taken to construct Crout-LU (label **Crout**), Doolittle-LU (label **Doolittle**), and REF-LU (label **REF**). Additionally, Table 5.2 provides a summary of run-times of the corresponding exact triangular solve routines on 50 RHS vectors. Both tables also report the ratios of the run-time taken by each of the rational arithmetic LU factorizations to the run-time taken by the REF LU factorization (labels **Crout/REF** and **Dool/REF**, respectively).

As Tables 5.1 and 5.2 demonstrate, the REF factorization framework achieved a decidedly superior performance than the exact rational arithmetic factorization approach. Its average construction run-times were up to 22 and 17 times faster as those achieved by Crout-LU and Doolittle-LU, respectively. Its relative performance reaches these peaks around $n = 250$ and then gradually decreases as n increases; this is likely due to the growing influence of slower-memory operations as the associated matrix entry bit-lengths become larger. Nonetheless, it is still able to factor a fully-dense 500×500 matrix—i.e., with 250,000 nonzeros—in under a minute while the exact rational arithmetic factorizations take over 15 minutes each. Even more impressively, the REF-LU average solution run-times were over 35 times faster than both Crout-LU and Doolittle-LU's across the board. The reported solution run-times

Table 5.1: Construction run-times (s): Crout-LU, Doolittle-LU, REF-LU

	Crout		**Doolittle**		**REF**		**Crout/REF**		**Dool/REF**	
n	AVG	SD	AVG	SD	AVG	SD	AVG	SD	AVG	SD
50	0.06	0.01	0.05	0.00	0.01	0.00	11.64	1.96	9.12	0.35
100	1.14	0.02	0.89	0.01	0.07	0.00	17.25	0.44	13.46	0.29
150	6.46	0.03	5.03	0.03	0.31	0.00	20.97	0.17	16.32	0.14
200	21.90	0.08	17.19	0.07	0.99	0.00	22.15	0.11	17.39	0.10
250	56.35	0.16	44.89	0.21	2.55	0.01	22.06	0.12	17.57	0.10
300	122.03	0.28	98.85	0.34	5.85	0.07	20.85	0.25	16.89	0.20
350	234.55	0.48	193.18	0.60	11.94	0.32	19.66	0.48	16.19	0.39
400	414.32	0.85	346.57	0.83	21.79	0.17	19.02	0.15	15.91	0.13
450	684.03	0.96	581.63	1.68	37.06	0.45	18.46	0.21	15.70	0.19
500	1073.25	1.49	924.51	3.15	59.67	0.37	17.99	0.11	15.49	0.09

Table 5.2: Solution run-times (s) of 50 RHS vectors: Crout-LU, Doolittle-LU, REF-LU

	Crout		**Doolittle**		**REF**		**Crout/REF**		**Dool/REF**	
n	AVG	SD	AVG	SD	AVG	SD	AVG	SD	AVG	SD
50	0.50	0.01	0.49	0.00	0.01	0.00	35.55	0.63	35.12	0.23
100	4.84	0.01	4.82	0.03	0.09	0.00	52.35	0.29	52.14	0.49
150	18.04	0.07	17.97	0.06	0.32	0.01	55.67	1.14	55.46	1.11
200	45.86	0.15	45.72	0.14	0.86	0.00	53.49	0.28	53.32	0.25
250	94.80	0.24	94.68	0.18	1.92	0.01	49.39	0.27	49.33	0.27
300	172.11	0.34	172.07	0.31	3.84	0.02	44.86	0.24	44.85	0.23
350	285.75	0.51	285.51	0.55	6.72	0.04	42.50	0.26	42.47	0.28
400	444.13	0.67	444.33	1.62	10.84	0.06	40.97	0.24	40.99	0.31
450	656.31	0.98	656.26	1.10	16.62	0.14	39.48	0.32	39.48	0.34
500	933.21	1.59	932.84	1.52	24.41	0.25	38.23	0.38	38.22	0.38

for $n = 500$ translate to approximately .5s per RHS vector for REF-LU and 19s for the exact rational arithmetic factorizations.

Among the two exact rational arithmetic LU factorizations, there was a virtual tie in the triangle solve run-times, which is to be expected based on the similarity of their contents (see Equations 5.3 and 5.8) and their use of the same standard forward

and backward substitution algorithms. However, Doolittle-LU outperformed Crout-LU in average construction run-times by more than 15% for every tested dimension value. This result seems to justify the preference for the Doolittle algorithm seen in theory and practice.

5.3.2.2 Storage

Having shown conclusive evidence of the computational advantages of implementing REF-LU over Doolittle-LU and Crout-LU, the second experiment examines the memory requirements for each of the factorizations. In particular, it performs a new set of runs to count and compare the number of whole limbs—i.e., 64-bit longs in the present computer environment—each factorization utilizes. The construction storage requirements are summarized in Table 5.3a in terms of the average number of limbs utilized by REF-LU, rounded to the nearest whole number, for each specified matrix dimension n (the standard deviation magnitudes were at most 0.5% of each average value and are thereby omitted for clarity). The average number of limbs utilized by Crout-LU and Doolittle-LU is expressed as a ratio of each corresponding REF-LU average value rounded to four decimal places. For these runs, only REF-LU is explicitly constructed. Doolittle-LU and Crout-LU are obtained by appending a denominator equal to a specific IPGE pivot to each REF-LU entry (see Equations (5.3) and (5.8)) and then reducing the resulting rational factorization entries to their lowest terms via gcd operations. As an added benefit, this shortcut makes it easy to determine what factors are removed from an original REF-LU entry to yield the smallest possible numerator in the corresponding entries of Crout-LU and Doolittle-LU (recall that the `GMP` library stores each rational entry in canonical form, meaning the numerator and denominator are relatively prime). These factors are of interest because they can be regarded as the memory cost of not having to store individual

denominators for REF-LU's entries. Table 5.3b summarizes these costs by recording the maximum such integer that is factored out of any REF-LU entry over the full set of repetitions for each value of n.

The second experiment also records three metrics related to the solutions obtained via the exact rational arithmetic approach (label **ERA**) and the REF framework: (1) solution size—i.e., the maximum bit-length of an individual entry in the solution vector—in limbs; (2) determinant bit-length in limbs; (3) number of instances in which the determinant must be explicitly stored as a denominator of an entry of the exact rational arithmetic solution vector. (In this part of the experiment, we do not differentiate between Doolittle-LU and Crout-LU since the process and outputs of solving an SLE via either exact rational arithmetic factorization are the same.) All the second experiment runs (from which the factorization storage metrics are concurrently collected) solve the SLE $A\boldsymbol{x} = \boldsymbol{e}_1$ for the purpose of uniformity, where $\boldsymbol{e}_1$ is the first n-length elementary vector. Hence, the RHS vector is constant, and the LHS A matrices are again different randomly-generated dense matrices. Each of these SLEs is a representative choice since it corresponds to solving for the first column of the inverse matrix; similar SLEs were solved in [75]. Table 5.4a and Table 5.4b summarize the results of the first metric and of the second and third metrics, respectively.

The results of the second experiment confirm earlier hypotheses regarding the storage relationship between REF-LU and the exact rational arithmetic factorizations. In particular, Table 5.3a shows that Crout-LU and Doolittle-LU utilize twice the number of average limbs asymptotically as REF-LU—i.e., as n increases, the respective average ratios relative to REF-LU approach two. Moreover, Table 5.3b reveals that the maximum factor removed from a REF-LU entry to transform it into the corresponding denominator of Crout-LU or Doolittle-LU was at most 10 digits

Table 5.3: Storage metrics: Crout-LU, Doolittle-LU, REF-LU

(a) Utilized Limbs

n	**REF-LU** AVG	**Crout/REF** AVG	**Dool/REF** AVG
50	6235	1.9583	1.9574
100	46571	1.9757	1.9753
150	155777	1.9822	1.9823
200	370251	1.9862	1.9862
250	727506	1.9890	1.9891
300	1265193	1.9908	1.9909
350	2022175	1.9920	1.9921
400	3038136	1.9929	1.9929
450	4351434	1.9937	1.9937
500	6002661	1.9944	1.9944

(b) Max Removed Factors

n	**Crout** MAX	**Doolittle** MAX
50	957783288	87071208
100	5853540	2862990
150	3357228	23500596
200	112150896	12628616
250	2810372084	6433882
300	26199300	55207425
350	16536240	25652700
400	270198825	56812032
450	7801242	20072132
500	34010361	121091232

Table 5.4: Solution metrics: ERA-LU, REF-LU

(a) Solution Size

	ERA Limbs		**REF Limbs**	
n	AVG	SD	AVG	SD
50	13.97	0.18	7.00	0.00
100	27.87	0.35	13.97	0.18
150	42.00	0.00	21.00	0.00
200	56.03	0.18	28.00	0.00
250	72.00	0.00	36.00	0.00
300	88.00	0.00	44.00	0.00
350	103.33	0.55	51.50	0.51
400	118.63	0.49	59.00	0.00
450	134.33	0.48	67.00	0.00
500	150.60	0.50	75.00	0.00

(b) Determinant Statistics

	Limbs		**Deteterminant = ERA Denom**
n	AVG	SD	
50	7.00	0.00	10 of 30 instances
100	14.00	0.00	8 of 30 instances
150	21.00	0.00	11 of 30 instances
200	28.13	0.35	13 of 30 instances
250	36.00	0.00	13 of 30 instances
300	44.00	0.00	17 of 30 instances
350	52.00	0.00	11 of 30 instances
400	59.73	0.45	15 of 30 instances
450	67.40	0.50	15 of 30 instances
500	75.73	0.45	11 of 30 instances

long accounting for the full set of runs involved in the experiment. Since a long can store 19 digits, in all of the aforementioned runs the numerators of the exact rational

arithmetic LU factorizations alone utilized exactly the same number of limbs as the full REF LU factorization. Thus, it seems that the storage cost paid by REF-LU to avoid keeping a denominator for each of its entries is insignificant relative to the substantial storage savings resulting from said avoidance.

Table 5.4a confirms the hypothesis that the solution sizes of the rational arithmetic LU factorizations are approximately twice as large as those required by the REF factorization framework. The rationale for said hypothesis comes from the fact that the result of REF forward and backward substitution is the integral vector $\boldsymbol{x}' = \text{Adj}(A)\boldsymbol{b}$, which is implicitly accompanied by the determinant of A as a denominator of every vector entry to yield the exact solution vector $\boldsymbol{x}$ (see Section 4.3). As it happens, the implicit common denominator comes essentially gratis to the REF substitution algorithms since the last diagonal element of REF-LU's lower-triangular matrix is equal to the determinant. This is significant because the average determinant had practically the same length as the REF-LU solution size as Table 5.4b reports and, more generally, because the exact rational arithmetic approach must store a separate denominator that may be as large as the determinant for each solution entry. In point of fact, as the third column of Table 5.4b shows, in at least 26% of the instances tested for each value of n, the exact rational arithmetic solution required the determinant to be explicitly stored as a denominator—i.e., at least one numerator in the exact solution was relatively prime to the determinant. Since the largest REF-LU bit-length usually corresponds to the basis determinant, this essentially indicates the REF factorization framework routinely stores the exact solution to $A\boldsymbol{x} = \boldsymbol{b}$ as compactly as possible.

5.4 Comparison with the Q-Matrix Revised Simplex Method

Azulay and Pique [7] designed the integer arithmetic-based Q-matrix revised simplex method to be a full-fledged exact LP solver. As Section 3.2.2 recounted, in their limited set of experiments, the method decreased the respective rational arithmetic revised simplex run-times by at most one order of magnitude—achieving a maximum speedup of 12 on one of 24 `NETLIB` instances, to be precise. Nevertheless, even if their method could sustain this performance edge, the savings would not suffice to overturn the conclusion reached from several implementations of exact rational arithmetic LP solvers (e.g., [17]), namely that deploying an unlimited-precision LP solver is impractical in most situations. For this reason, the next subsection first derives an adaptation of the Q-matrix method that enables its efficient implementation as a basic solution validation subroutine. Afterward, Section 5.4.2 compares the REF factorization framework with respect to this streamlined version of the Q-matrix method.

5.4.1 A Minimal Implementation of Q-Matrices

The Q-matrix revised simplex method works by constructing the adjunct matrix of a basis and then by updating said matrix via Q-pivots and IPGE variants of the traditional revised simplex operations. To validate a basic solution, however, it is unnecessary to perform individual operations within the simplex algorithm. In this featured context, therefore, the only essential parts of the Q-matrix revised simplex method are the construction of the adjunct matrix from a predetermined set of column indices and its implementation in calculating an exact SLE solution. Assuming A is the basis matrix to be validated, the first part can be accomplished in $O(n^3)$ operations by applying the full version of IPGE to the augmented matrix $[A|I_n]$, so that its LHS becomes a diagonal matrix with all its nonzero entries equal

to $\det(A)$. As anticipated, $\text{Adj}(A)$ emerges as the accompanying right-hand matrix. After this initial construction, the second essential part of the Q-matrix framework is easily performed in $O(n^2)$ operations by a matrix-vector multiplication involving $\text{Adj}(A)$ and a RHS vector $\boldsymbol{b}$ (in the case of the primal solution calculation).

Based on the preceding discussion, it is necessary to allocate an $n \times 2n$ matrix for the construction of $\text{Adj}(A)$, but only an $n \times n$ matrix during the exact solution calculation. Next, we introduce an alternative construction process requiring an $n \times n$ matrix as well, thereby reducing the previous storage requirement by one-half. The algorithm is described by the following equation, which needs to be evaluated for $k = 1 \ldots n$, $i = 1 \ldots n$, and $j = 1 \ldots k\text{–}1, k\text{+}1 \ldots n, k$ (in this order):

$$a_{i,j}^{k} = \begin{cases} \rho^{k-1} & \text{if } i = k, j = k \\ -a_{i,j}^{k-1} & \text{if } i \neq k, j = k \\ a_{i,j}^{k-1} & \text{if } i = k, j \neq k \\ (\rho^{k} a_{i,j}^{k-1} - a_{k,j}^{k-1} a_{i,k}^{k-1})/\rho^{k-1} & \text{if } i \neq k, j \neq k; \end{cases} \tag{5.9}$$

where $a_{i,j}^{0} = a_{i,j}$. The algorithm is identical to the full version of IPGE specified by Equation (2.7), except that the kth pivot column, $A_{(:,k)}^{k-1}$, is overwritten by itself times negative one in rows $i \neq k$ or by the previous pivot, ρ^{k-1}, in row k. Based on the above expressions, it is straightforward to perceive that the algorithm performs $O(n^3)$ operations and that it requires storing only an $n \times n$ matrix.

In what follows, Lemma 2 will provide the rationale for overwriting the pivot column at each step with the values indicated in Equation (5.9); afterward, Theorem 5.4.1 proves the above algorithm is correct and REF. To this end, let $A_{(:,n+j)}^{k}$ denote the kth-iteration jth RHS column obtained when applying IPGE—according to the sequence of pivot coordinates designated by Assumption 2—on $[A|I_n]$, where $1 \leq$

$j, k \leq n$.

Lemma 2. *Provided that the* $(k\text{–}1)$*th-iteration RHS column,* $A^{k-1}_{(:,n+j)}$*, equals* $\rho^{k-1}\boldsymbol{e}_j$*, for* $k > 0$*, the next iterate of the same column,* $A^{k}_{(:,n+j)}$*, can be obtained via the following shortcuts:*

$$A^{k}_{(:,n+j)} = \begin{cases} -A^{k-1}_{(:,k)} + (\rho^k + \rho^{k-1})\boldsymbol{e}_k & \text{if } j = k \\ \rho^k \boldsymbol{e}_j & \text{otherwise;} \end{cases}$$

where $\boldsymbol{e}_j$ *is the* n*-length* j*th elementary vector,* ρ^k *is the* k*th IPGE pivot element (equal to* $a^{k-1}_{k,k}$ *by Assumption 2), and* $\rho^0 = 1$*.*

Proof. The proof is divided into two parts. When $j = k$, applying the definition of IPGE to the calculation of the individual entries of $A^{k}_{(:,n+j)}$ gives:

$$a^{k}_{i,n+k} = \begin{cases} a^{k-1}_{i,n+k} = a^{k-1}_{k,n+k} & = \rho^{k-1} \quad \text{if } i = k \\ (\rho^k a^{k-1}_{i,n+k} - a^{k-1}_{k,n+k} a^{k-1}_{i,k})/\rho^{k-1} = (\rho^k(0) - \rho^{k-1} a^{k-1}_{i,k})/\rho^{k-1} & = -a^{k-1}_{i,k} \quad \text{otherwise,} \end{cases}$$

where ρ^{k-1} is substituted in place of $a^{k-1}_{k,n+k}$ according to the given assumption. Notice that, since $a^{k-1}_{k,k} = \rho^k$, the $i = k$ case above can be equivalently expressed as:

$$a^{k}_{k,n+k} = -a^{k-1}_{k,k} + (a^{k-1}_{k,k} + \rho^{k-1}) = -a^{k-1}_{k,k} + (\rho^k + \rho^{k-1}),$$

thus completing the first part of the proof. Similarly, for $j \neq k$, starting with the IPGE definition yields:

$$a^{k}_{i,n+j} = \begin{cases} a^{k-1}_{i,n+j} & = 0 \quad \text{if } i = k \\ (\rho^k a^{k-1}_{i,n+j} - a^{k-1}_{k,n+j} a^{k-1}_{i,k})/\rho^{k-1} = (\rho^k \rho^{k-1} - (0) a^{k-1}_{i,k})/\rho^{k-1} & = \rho^k \quad \text{if } i = j \\ (\rho^k a^{k-1}_{i,n+j} - a^{k-1}_{k,n+j} a^{k-1}_{i,k})/\rho^{k-1} = (\rho^k(0) - (0) a^{k-1}_{i,k})/\rho^{k-1} & = 0 \quad \text{otherwise.} \end{cases}$$ □

Theorem 5.4.1. *The algorithm specified by Equation* (5.9) *calculates the adjunct matrix Adj*(A) *without accruing roundoff errors.*

Proof. Since the iterative LHS matrix obtained by applying IPGE on $[A|I_n]$ is only needed to transform I_n into Adj(A), columns $A^{k-1}_{(:,1)}, \ldots, A^{k-1}_{(:,k-1)}$ are superfluous during the algorithm's kth to nth iterations (in fact, at iteration k, they could all be retrieved via the shortcut given by Equation (2.8)). Hence, after completing the kth iteration, column $A^k_{(:,k)}$ will also become superfluous, meaning a new column can be inserted in place of it without losing any critical data. The following analysis explains that the inserted column corresponds to $A^k_{(:,n+k)}$ and that only the first k RHS columns need to be explicitly stored up to this point.

Continuing, since $\rho^0 = 1$, the n-order identity matrix can be equally characterized as:

$$I_n = \rho^0 I_n = [\rho^0 \boldsymbol{e}_1 \;\; \rho^0 \boldsymbol{e}_2 \;\; \ldots \;\; \rho^0 \boldsymbol{e}_n].$$

This implies that Lemma 1 can be applied in the first IPGE iteration on all the RHS columns of $[A^0|I_n]$ to obtain $A^1_{(:,n+1)}, \ldots, A^1_{(:,2n)}$ via its shortcuts. In the second IPGE iteration, Lemma 1 will apply to all but the first of these columns since $A^1_{(:,n+2)} = \rho^1 \boldsymbol{e}_2, \ldots, A^1_{(:,2n)} = \rho^1 \boldsymbol{e}_n$, but $A^1_{(:,n+1)} = -A^0_{(:,n+1)} + (\rho^1 + \rho^0)\boldsymbol{e}_1 \neq \rho^1 \boldsymbol{e}_1$. Propagating this simple argument, in iteration k the shortcuts can be applied only to columns $A^{k-1}_{(:,n+k)} = \rho^{k-1}\boldsymbol{e}_k, \ldots, A^{k-1}_{(:,2n)} = \rho^{k-1}\boldsymbol{e}_n$. More importantly, since the content of these columns is readily deducible according to Lemma 2, only the first k RHS columns need to be explicitly stored following this iteration. As it happens, they can occupy exactly the first k columns of the LHS matrix since the columns $A^k_{(:,1)}, \ldots, A^k_{(:,k)}$ are superfluous at this point, as the above paragraph explained. Thus, by first performing the usual IPGE operations on the kth-iteration non-pivot columns and then generating column $A^k_{(:,n+k)}$ to replace the pivot column $A^{k-1}_{(:,k)}$ via Lemma 2, for

$k = 1$ to n, the algorithm will yield $\text{Adj}(A)$ as the output LHS matrix. In other words, there is no need to store the RHS matrix at any point. Lastly, since this algorithm is simply a storage-efficient version of the normal application of IPGE on $[A|I_n]$, all its calculations are free of roundoff errors. □

Having developed and proved the correctness of the above-mentioned algorithm, the Q-matrix revised simplex method can now be efficiently implemented and compared in the same context and leveled field as the REF LU factorization. In particular, like REF-LU, $\text{Adj}(A)$ can be computed in $O(n^3)$ operations without roundoff errors using only an $n \times n$ matrix, after which it can be deployed to calculate the exact solution to $A\boldsymbol{x} = \boldsymbol{b}$ in $O(n^2)$ operations, for any RHS vector $\boldsymbol{b}$. Instead of performing two triangular solves, however, the Q-matrix revised simplex method calculates the exact primal solution via the following matrix-vector multiplication:

$$\boldsymbol{x}' = \text{Adj}(A)\boldsymbol{b} = \left[\det(A)A^{-1}\right]\boldsymbol{b} = \det(A)\boldsymbol{x};$$

where the second equality results from the definition of the adjunct matrix and the third from Cramer's rule. Notice that the multiplication produces the same scaled vector $\boldsymbol{x}'$ as the end result of REF forward and backward substitution. Therefore, the exact solution to $A\boldsymbol{x} = \boldsymbol{b}$ is stored as:

$$x_i = \frac{x'_i}{\det(A)} \qquad \text{for } i = 1 \ldots n.$$

Recalling that the adjunct matrix of A is defined as its transpose cofactor matrix,

$\mathrm{Adj}(A)$ can be expressed in terms of subdeterminants of A as follows:

$$\mathrm{Adj}(A) = \begin{bmatrix} \alpha_{N\setminus\{1\}}^{N\setminus\{1\}} & -\alpha_{N\setminus\{1\}}^{N\setminus\{2\}} & \alpha_{N\setminus\{1\}}^{N\setminus\{3\}} & \cdots & -\alpha_{N\setminus\{1\}}^{N\setminus\{n\}} \\ -\alpha_{N\setminus\{2\}}^{N\setminus\{1\}} & \alpha_{N\setminus\{2\}}^{N\setminus\{2\}} & -\alpha_{N\setminus\{2\}}^{N\setminus\{3\}} & \cdots & \alpha_{N\setminus\{2\}}^{N\setminus\{n\}} \\ \alpha_{N\setminus\{3\}}^{N\setminus\{1\}} & -\alpha_{N\setminus\{3\}}^{N\setminus\{2\}} & \alpha_{N\setminus\{3\}}^{N\setminus\{3\}} & \cdots & -\alpha_{N\setminus\{3\}}^{N\setminus\{n\}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\alpha_{N\setminus\{n\}}^{N\setminus\{1\}} & \alpha_{N\setminus\{n\}}^{N\setminus\{2\}} & -\alpha_{N\setminus\{n\}}^{N\setminus\{3\}} & \cdots & \alpha_{N\setminus\{n\}}^{N\setminus\{n\}} \end{bmatrix}, \tag{5.10}$$

where $N = \{1, \ldots, n\}$ is the column index set and n is even (solely for the purposes of this illustration). Based on this simple fact, the bit-length of *every* entry of $\mathrm{Adj}(A)$ has the bound $(n-1)\log(\sqrt{(n-1)\sigma})$, based on Hadamard's determinant inequality (see Equations (2.13) and (2.14)). On the other hand, the entries along row i of U and column i of L in REF-LU have bit-lengths with the bound $i\log(\sqrt{i\sigma})$ also due to Hadamard's determinant inequality (i.e., see the expression of REF-LU's L and U matrices as subdeterminants of A given by Equation (5.5)). This indicates that REF-LU requires significantly less storage than $\mathrm{Adj}(A)$.

5.4.2 Experiments and Results

The present subsection reports and analyzes the results of computational tests designed to compare the REF factorization framework with the adapted Q-matrix approach for basic solution validation. The two main experiments developed for this purpose are in essence the analogs of the experiments described in Sections 5.3.2.1 and 5.3.2.2.

5.4.2.1 Run-Times

The first experiment consists of measuring the run-times required to construct the core computational tools of the REF factorization framework and of the Q-matrix

approach and the run-times of applying each tool to solve an SLE exactly. The respective computational tools are REF-LU and the adjunct matrix of A derived via the adapted Q-matrix approach (hereafter alternatively referred to as Q-Mat-Adj). For this experiment, the dense input matrix dimension range is amplified to $n \in [100, 1000]$; accordingly, the largest input matrices tested have exactly 1,000,000 non-zeros. Table 5.5 provides a summary of the run-times in seconds (s) taken to construct REF-LU and Q-Mat-Adj for the input A matrices. It also reports the ratios of the construction run-time taken by Q-Mat-Adj to the run-time taken by REF-LU (label **Q-Mat/REF**). Additionally, Table 5.6 provides a summary of the run-times taken to perform the REF-LU-based triangular solves or to perform matrix-vector multiplications—i.e., the SLE solution subroutine for the Q-matrix approach—for 50 RHS vectors. For this metric, the table reports the ratios of the solution run-time taken by the REF-LU to the run-time taken by Q-Mat-Adj (label **REF/Q-Mat**).

Notice that REF-LU achieved the best construction run-times once again, this time outperforming Q-Matrix-Adj by close to a factor of nine. Moreover, it appears this performance edge would continue its upward trend if higher values of n were tested. Q-Matrix-Adj manages to perform twice as fast as the exact rational arithmetic LU factorizations, as attested by a quick cross-reference of Tables 5.1 and 5.5 along their overlapping dimension values. More notably, Q-Mat-Adj beats REF-LU in terms of solution run-times by upwards of a factor of 22, which also appears to rise for higher values of n than those tested. This reversal in relative performance is to be expected based on the respective exact SLE solution subroutines required by each method. Expressly, the Q-Mat-Adj solution procedure simply computes an integer matrix-vector multiplication, while REF-LU performs REF forward and backward substitution, which includes carrying out exact integer divisions.

Table 5.5: Construction run-times (s): Q-Mat-Adj, REF-LU

	Q-Mat-Adj		**REF-LU**		**Q-Mat/REF**	
n	AVG	SD	AVG	SD	AVG	SD
100	0.33	0.02	0.07	0.00	5.05	0.20
200	7.27	0.03	0.99	0.04	7.33	0.23
300	45.50	0.21	5.86	0.05	7.76	0.06
400	172.76	0.38	21.91	0.14	7.88	0.04
500	486.61	2.49	60.53	0.32	8.04	0.04
600	1143.49	3.95	139.75	1.12	8.18	0.06
700	2367.04	12.10	283.59	1.52	8.35	0.05
800	4468.37	16.56	525.30	2.50	8.51	0.04
900	7933.58	82.07	916.87	13.41	8.65	0.10
1000	13279.76	107.65	1508.10	22.52	8.81	0.12

Table 5.6: Solution run-times (s) of 50 RHS vectors: Q-Mat-Adj, REF-LU

	Q-Mat-Adj		**REF-LU**		**REF/Q-Mat**	
n	AVG	SD	AVG	SD	AVG	SD
100	0.03	0.00	0.09	0.00	2.71	0.04
200	0.18	0.00	0.86	0.03	4.72	0.19
300	0.49	0.01	3.86	0.04	7.84	0.24
400	1.13	0.01	10.83	0.04	9.56	0.11
500	2.08	0.01	24.43	0.20	11.76	0.10
600	3.34	0.04	47.61	0.37	14.24	0.17
700	5.06	0.02	83.23	0.41	16.46	0.09
800	7.34	0.11	135.67	0.65	18.49	0.27
900	10.46	0.27	211.73	2.44	20.25	0.46
1000	14.15	0.25	313.99	3.53	22.20	0.41

5.4.2.2 Storage

The second experiment examines the memory requirements of Q-Mat-Adj and REF-LU. The construction storage requirements of REF-LU are summarized in Table 5.7a using the average number of limbs, rounded to the nearest whole number, for each specified matrix dimension n (the standard deviation magnitudes are omitted

due to their relatively insignificant magnitude). The average construction requirements of Q-Mat-Adj are expressed as ratios of the corresponding REF-LU average values (label **Q-Mat/REF**) rounded to four decimal places. As Section 5.4.1 explains, the respective exact SLE solution subroutines of both methods yield the scaled solution vector $\boldsymbol{x}' = \text{Adj}(A)\boldsymbol{b}$ associated with IPGE. For this reason, the IPGE solution size—i.e., the maximum bit-length of an individual entry in $\boldsymbol{x}'$—displayed in Table 5.7b encompasses both approaches; the table reports the average solution sizes and their standard deviations in terms of bit-length and limbs. We remark that the RHS vector $\boldsymbol{b}$ is fixed as $\boldsymbol{b} = \boldsymbol{e}_1$ for the purpose of uniformity as in the analogous computational tests described in Section 5.3.2.2.

The results of Table 5.7a confirm the hypothesis that the adjunct matrix of the basis requires significantly more storage than the REF LU factorization. In particular, Q-Mat-Adj requires approximately three times the number of limbs as REF-LU, and this value appears to rise gradually with n. We expect that REF-LU's advantage in this respect would be even higher for sparse matrices since LU factorizations can be designed to minimize fill-in while the inverse matrix is generally dense [33]. Notice also that, based on the results reported in Section 5.3.2.2, Q-Mat-Adj actually requires over 50% more storage than the exact rational arithmetic LU factorizations. The results of additional computational tests in Section 5.4.2.3 look at how these elevated memory requirements impact the relative computational performances of both alternative approaches as the input matrix size increases.

Table 5.7b leads to several key observations. For starters, stating the solution size in terms of bit-length establishes a connection with the computational tests performed in [17]. In said study, which Section 5.1 describes in brief, solution size is used as one of three attributes for grouping the tested instances. It is shown therein that when the solution size is large—which is roughly characterized as exceeding

Table 5.7: Storage metrics: Q-Mat-Adj, REF-LU, IPGE Solution

(a) Utilized Limbs

	REF-LU	**Q-Mat/REF**
n	AVG	AVG
100	46585	2.9610
200	370236	3.0251
300	1265339	3.1292
400	3037594	3.1077
500	6003518	3.1232
600	10482385	3.1593
700	16797440	3.1505
800	25280548	3.1645
900	36262807	3.1496
1000	50076523	3.1552

(b) IPGE Solution Size

	Bit-length		**Limbs**	
n	AVG	SD	AVG	SD
100	837.27	2.89	13.93	0.25
200	1782.37	1.61	28.00	0.00
300	2764.03	2.62	44.00	0.00
400	3770.13	2.42	59.00	0.00
500	4795.43	2.71	75.03	0.18
600	5835.30	2.59	92.00	0.00
700	6887.43	2.24	108.00	0.00
800	7948.97	1.96	125.00	0.00
900	9020.70	2.45	141.07	0.25
1000	10100.07	2.82	158.00	0.00

1,000 bits—the average computational performance of the exact rational arithmetic LU factorization approach is better than that of p-ading lifting and Wiedemann's method; it is reasonably close to the top average performance achieved by iterative refinement (which is also the only approach that failed to solve five instances in the alluded study). Since the exact rational arithmetic solution vector tends to require nearly twice the IPGE solution size in limbs (see Table 5.4a), this implies the solution of the SLE $A\boldsymbol{x} = \boldsymbol{e}_1$ for all our instances yields large rational solution sizes. To test this relationship in terms of bit-length, we divided each IPGE solution vector $\boldsymbol{x}'$ by $\det(A)$ to yield $\boldsymbol{x}$, and then reduced each resulting rational to canonical form and measured its bit-length. As expected, the average solution size of $\boldsymbol{x}$ was 2.00 for each value of n. Hence, based on the reported averages and relatively minor standard deviation values, none of the solution vectors appears to be small in size. It is important to mention that the basis determinant, which implicitly accompanies

the IPGE solution to obtain $\boldsymbol{x}$ from $\boldsymbol{x}'$, is available as the last diagonal of REF-LU but is not present in Q-Mat-Adj (it is obtained during its construction, but must be stored separately).

5.4.2.3 Additional Computational Tests

This subsection describes the results an additional experiment designed to analyze further the computational performance of Doolittle-LU, Q-Mat-Adj, and REF-LU. Tables 5.8 and 5.9 summarize the results of the extended construction and solution tests, respectively; the associated tests replicate the computational tests done in Sections 5.3.2.1 and 5.4.2.1 with the distinction that the chosen dimension parameters are powers of 2 rather than multiples of 50. Specifically, the tested dimensions are $n = 2, 4, \ldots, 2048$ for REF-LU and $n = 2, 4, \ldots, 1024$ for Doolittle-LU and Q-Mat-Adj; the $n = 2048$ setting—i.e., the input matrix with exactly 4,194,304 non-zeros—was not tested for Doolittle-LU and Q-Mat-Adj since the respective construction processes (on which the solution subroutines also depend) of each instance are projected to take approximately one week in each case. On a related note, due to the longer run-times expected in this experiment, the corresponding repetitions were performed on 15 separate machines with the specifications described in Section 5.2.1; each of the experiments described in previous subsections performed its full set of repetitions on a single machine (although the individual machines were likely distinct for each experiment). Owing to the distribution of experiment repetitions over separate machines, the standard deviation values are larger than those previously observed, though they remain relatively small. As in previous experiments, the solution algorithms were applied to 50 RHS vectors. However, in order to provide estimates of the run-times required to solve one RHS vector at a time, Table 5.9 reports the 50-vector averages and standard deviations divided by 50; the values are

rounded to four decimal places to account for the lower magnitudes associated with smaller basis dimensions.

Table 5.8: Extended construction run-times (s): Doolittle-LU, Q-Mat-Adj, REF-LU

	Doolittle-LU		**Q-Mat-Adj**		**REF-LU**		**Dool/REF**	**Q-Mat/REF**
n	AVG	SD	AVG	SD	AVG	SD	AVG	AVG
64	0.17	0.04	0.08	0.03	0.02	0.00	7.66	3.65
128	2.66	0.04	0.96	0.04	0.18	0.02	15.06	5.46
256	50.62	0.85	22.79	1.77	2.89	0.22	17.53	7.89
512	1036.81	3.56	554.05	23.42	67.78	3.87	15.30	8.17
1024	22812.61	184.43	14938.05	107.67	1697.77	158.55	13.44	8.80
2048	–	–	–	–	44025.15	1493.82	–	–

Figure 5.1: Log-log plot of Doolittle-LU, Q-Mat-Adj, and REF-LU extended construction run-times

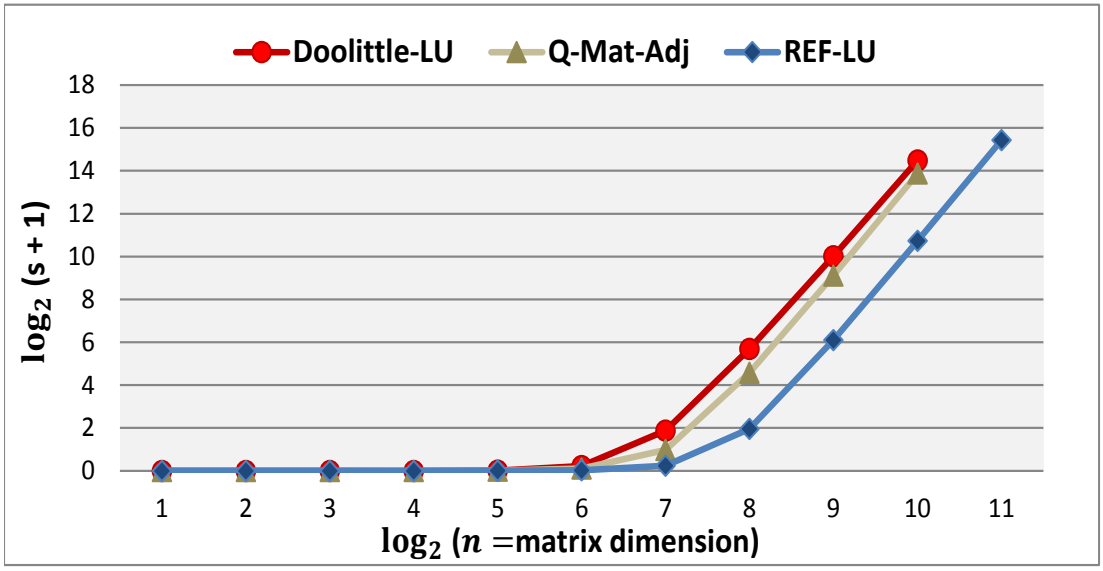
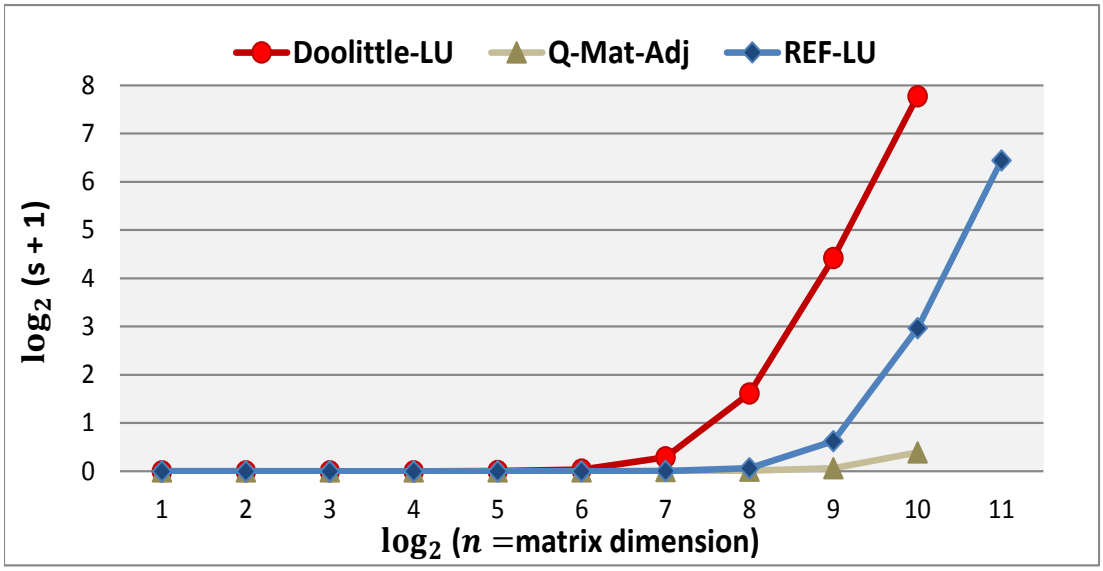


Table 5.9: Extended solution run-times (s) per RHS vector (50-vector values divided by 50): Doolittle-LU, Q-Mat-Adj, REF-LU

	Doolittle-LU		**Q-Mat-Adj**		**REF-LU**		**Dool/REF**	**Q-Mat/REF**
n	AVG/50	SD/50	AVG/50	SD/50	AVG/50	SD/50	AVG	AVG
64	0.0227	0.0003	0.0003	0.0001	0.0007	0.0002	34.0590	0.4240
128	0.2200	0.0096	0.0012	0.0000	0.0041	0.0006	53.3990	0.3003
256	2.0656	0.0072	0.0070	0.0004	0.0424	0.0056	48.7419	0.1662
512	20.3742	0.0631	0.0437	0.0004	0.5366	0.0401	37.9715	0.0814
1024	217.9290	1.3254	0.3091	0.0080	6.8175	0.2021	31.9660	0.0453
2048	–	–	–	–	86.1521	6.2144	–	–

Figure 5.2: Log-log plot of Doolittle-LU, Q-Mat-Adj, and REF-LU extended solution run-times per RHS vector



The experiment results hint at the asymptotic relationships between the run-times of Doolittle-LU, Q-Mat-Adj, and REF-LU. In order to bring these relationships to light, Figures 5.1 and 5.2 provide log-log plots of the average construction run-times reported in Tables 5.8 and 5.9, respectively, in seconds (s). To be precise, the plotted values are the base-2 log average run-times (shifted by one second) corresponding

to the base-2 log of each value of n. From this graph, it is possible to discern the point at which slower-memory operations begin to have a substantial effect on each corresponding construction algorithm. Indeed, after $n = 2^8$, the performance of each of the three approaches is roughly characterized by a distinct power function (i.e., having a distinct but fixed exponent). However, the slowdown occurs sooner for Doolittle-LU and then Q-Mat-Adj since they require between two and three times the memory of REF-LU as Sections 5.3.2.2 and 5.4.2.2 explain. Figure 5.1 also illustrates the gradually increasing ratio of the REF-LU construction run-times relative to those of Q-Mat-Adj. A similar but reversed relationship between the solution run-times of both approaches can be observed in Figure 5.2; REF-LU performs significantly better than Doolittle-LU, which achieves the worst performance in this respect. Most interestingly, based on the gradually declining performance of Q-Mat-Adj seen in Figure 5.1, it appears that Doolittle-LU would eventually prevail over Q-Mat-Adj in terms of construction run-times for large n. This is likely due to the costs associated with the growing memory requirements of Q-Mat-Adj outweighing the costs associated with Doolittle-LU's rational arithmetic operations in the long run.

5.5 Conclusions

This section demonstrates that the REF factorization framework outperforms the exact rational arithmetic LU factorization approach, which is the fail-safe tool utilized by both state-of-the-art exact mathematical programming solvers `QSopt_ex` and `SoPlex` to solve SLEs exactly. In particular, the featured experiments demonstrate that the REF factorization algorithms are significantly faster than their exact rational arithmetic counterparts. Indeed, the factors of improvement are as high as 22 and 55 for the factorization constructions and triangular solves, respectively.

Moreover, the REF LU factorization uses virtually half the memory required by the exact rational arithmetic Doolittle and Crout factorizations. This finding confirms the insights arising from the theoretical derivations and analysis conducted earlier in the work connecting the three exact factorizations. This section also develops and tests an adaptation of the Q-matrix revised simple method that can be efficiently applied toward basic solution validation. Related experiments reveal that the REF LU factorization can be constructed close to nine times faster than the Q-matrix-based adjunct matrix for the larger tested instances. However, these experiments also show that the Q-matrix SLE solution can be more than 22 times faster than REF forward and backward substitution algorithms over the same larger instances. Nevertheless, based on the various considerations to be discussed in the next paragraph, the REF factorization framework remains the preferred choice among all the tested exact SLE solution methods.

The REF factorization framework is preferable to the Q-matrix approach in the context basic solution validation owing to four key considerations. First, construction run-times are significantly greater in magnitude than solution run-times for each method; in the featured experiments this magnitude difference widens as n increases, which is explained by the fact that the construction and solution subroutines of both approaches require $O(n^3)$ and $O(n^2)$ operations, respectively. Second, the average run-times displayed in Table 5.6 actually correspond to the solution of 50 RHS vectors at each repetition. Validating a basic solution or even determining the entering column in the simplex algorithm usually involves solving for only one RHS vector using the same basis and, thus, the aforementioned solution run-times are a significant overrepresentation of the efforts that would be associated with solving the featured instances. The results reported in Table 5.9 provide closer estimates of the solution run-times involved in each approach. Third, the experiment results

appear to indicate that, based on its growing memory requirements, the Q-matrix adjunct matrix would eventually take longer to construct than the exact rational arithmetic LU factorizations for larger matrix dimensions. Fourth, we project that a similar comparison on sparse instances would improve the performance of the REF factorization framework considerably more than that of the Q-matrix approach. The reason is that, generally speaking, LU factorization algorithms can be designed to preserve and exploit the sparsity of an SLE, in contrast to the construction and deployment of the adjunct matrix, which is a scaled version of the usually dense inverse matrix. Hence, the tested dense matrices are those in which the REF LU factorization cannot profit from this significant advantage. In future work, the hypothesized accelerated performance of the REF factorization framework for sparse instances will be explicitly tested.

As a final remark, we predict that the computational performance of every method herein tested would be proportionally enhanced by the use of the `EGlib` library [29] for fast memory allocation and by other technical advancements to our codes. Hence, it is highly likely that the reported performance of each tested unlimited-precision method could be improved fairly evenly in more sophisticated implementations. For this reason the computational performance realized by the REF factorization framework can be regarded as a starting point for prospective finely tuned implementations.

6. ROUNDOFF-ERROR-FREE BASIS UPDATES OF LU FACTORIZATIONS FOR EFFICIENT VALIDATION OF OPTIMALITY CERTIFICATES

6.1 Introduction

Ever since Bartels and Golub published their landmark paper in [10], LU factorization algorithms have been continually improved with regard to their implementation within linear programming (LP). In particular, Bartels and Golub [10] demonstrated that the application of LU factorization for solving SLEs within the simplex method was efficient and more numerically stable than the then-popular product form of the inverse approach. Subsequent papers thenceforth enhanced the two highlighted attributes of the Bartels-Golub method through the development and recurrent improvement of LU and Cholesky factorization update algorithms—e.g., [69, 32, 68, 36, 79, 67].

Owing to these and several other noteworthy advances, LU and Cholesky factorizations have become a common feature of most solvers; yet they, along with their associated algorithms, are still susceptible to nontrivial roundoff errors [49]. As Section 1 explains, these errors may lead mathematical programming solvers to provide invalid output. The possibility that these incorrect solver conclusions are being used to make critical real-world decisions is disconcerting. Consequently, this underscores the importance of continuing to develop computationally viable factorization algorithms that incur even less roundoff error or that eliminate it altogether.

In order to address this key issue, Section 4 (i.e., [27]) develops the REF LU and Cholesky factorizations as well as REF forward and backward substitution algorithms. The REF computational tools stem from IPGE, and they share the property that their individual coefficients' bit-lengths (i.e., required number of storage bits)

are bounded polynomially. And, unlike the exact rational arithmetic LU factorizations used by state-of-the-art exact LP solvers—e.g., `QSopt_ex` [5] and `SoPlex` [88, 89]—as detailed in Section 3, the REF factorization algorithms do not require any gcd calculations to achieve this polynomial bound. Hence, their prospective implementation in exact LP would have the benefit of saving the high costs associated with these operations. However, before the extent of this conjecture can be adequately verified, it is necessary to develop appropriate efficient algorithms for updating the REF factorizations. Thus, this section extends the applicability of the REF factorization framework by developing algorithms that efficiently update the REF LU and Cholesky factorizations when various types of modifications are performed on the underlying SLE. This research direction is necessary because, as Section 6.3.1 explains, applying the traditional insert-delete-reduce approach to update the REF factorizations is inefficient in terms of operand bit-length growth and increased computational effort. The algorithms herein developed differ significantly from their traditional counterparts in that they follow a push-and-swap update approach, which preserves the special structure of the REF factorizations.

This section makes the following contributions. First, the current work concludes that applying the traditional delete-insert-reduce approach to update the REF factorizations can be costlier than constructing the corresponding REF factorizations from scratch. Second, it develops the push-and-swap approach for updating the REF factorizations efficiently and without accruing roundoff errors. The featured update operations are column addition, deletion, and replacement; in addition, this work also proves that the complementary row updates can be performed via the column updates. Third, it proves that these operations achieve the computational savings traditionally expected of factorization updates and that they do not lead to further growth in the bit-length of their matrix entries. Fourth, it introduces a set of

REF operations that augment the versatility of IPGE and the REF factorization framework.

This section is organized as follows. Section 6.2 provides an overview of the REF factorization framework for solving SLEs through a stylized matrix structure termed a frame matrix. Section 6.3 summarizes the conventional and proposed factorization update approaches in relation to the REF LU factorization: Section 6.3.1 explains the inadequacy of applying the traditional delete-insert-reduce approach, and Section 6.3.2 outlines the proposed frame matrix-based push-and-swap update methodology. Section 6.4 broadens the application of the REF framework: Section 6.4.1 defines and proves the correctness of auxiliary IPGE-type operations, and 6.4.2 combines them to form a set of elementary updates that serve as building blocks of the featured update algorithms. Section 6.5 develops and proves the correctness and complexity of the REF column addition, deletion, and replacement updates for the REF LU factorization. Section 6.6 provides a step-by-step depiction of the REF update process via a numerical example that helps reinforce many of the concepts herein introduced; as such, this section will periodically refer to specific parts of it to supplement the discussion. Section 6.7 proves that the corresponding row updates can be performed via the featured column updates, and it discusses special considerations for updating the Cholesky factorization. Lastly, Section 6.8 concludes the section and discusses future related research efforts.

6.2 The REF Factorization Framework via Frame Matrices

The REF factorization framework is composed of the REF LU and Cholesky factorizations and of the REF forward and backward substitution algorithms, all of which are founded on the application of IPGE. A traditional definition of IPGE is given in Section 2.3.5.1 and a generalized definition is introduced in Section 6.4.

In order to expand the REF factorization framework, descriptions of the REF LU factorization and the REF substitution algorithms via a stylized matrix structure termed a frame matrix are presented next. To aid with the present discussion, recall that when applying IPGE, ρ^k denotes the kth pivot element selected from the $(k–1)$th iterative matrix, $A^{k–1}$, to perform the algorithm's kth iteration; the row and column coordinates of ρ^k are represented by the ordered pair (r_k, c_k). For the same purpose, it is expedient to restate two previous assumptions from Section 2, the second of which is to be removed later in this section.

Assumption 1. *Let $A\boldsymbol{x} = \boldsymbol{b}$ be a nonsingular SLE with coefficient matrix $A \in \mathbb{Z}^{n\times n}$, right-hand side vector $\boldsymbol{b} \in \mathbb{Z}^n$, and variable vector $\boldsymbol{x} \in \mathbb{Q}^n$.*

Assumption 2 (temporary)**.** *Fix $\rho^k = a_{k,k}^{k-1} \neq 0$ (i.e., $r_k = c_k = k$), for $k \geq 1$. (Starting in Section 6.4, this assumption will be removed.)*

We remark that, based on Assumption 2, it is not necessary to perform any row or column permutations between iterations of a Gaussian elimination algorithm; equivalently, each corresponding permutation matrix is the identity matrix.

The SLE $A\boldsymbol{x} = \boldsymbol{b}$ can be solved without accruing roundoff errors utilizing the REF factorizations and the REF substitution algorithms developed in Section 4. Without loss of generality, the ongoing narrative centers on the REF LU factorization (REF-LU) since its L and D matrices are equal to their counterparts in the REF Cholesky factorization and since $U = L^T$ when A is symmetric positive definite.

Let $i, j \in \mathbb{Z}$ such that $1 \leq i, j \leq n$. Recall that the nonzero entries of each matrix

factor in REF-LU equate with individual IPGE entries as follows:

$$\begin{aligned} l_{i,j} &= a_{i,j}^{j-1}, && \text{for } i \geq j; \\ d_{i,i} &= \rho^{i-1}\rho^{i} = a_{i-1,i-1}^{i-2} a_{i,i}^{i-1}, && \text{for all } i; \text{ and} \\ u_{i,j} &= a_{i,j}^{i-1}, && \text{for } i \leq j. \end{aligned}$$

Notice that D's entries can be generated on the fly since $a_{i-1,i-1}^{i-2} a_{i,i}^{i-1} = l_{i-1,i-1} l_{i,i} = u_{i-1,i-1} u_{i,i}$, where $a_{0,0}^{-1} := \rho^0 = 1$. Moreover, the overlap between the diagonal elements of L and U suggests that these two matrices can be merged into a single $n \times n$ matrix. We denote this single matrix as the *frame matrix* of A, or F, for short. In particular, with the diagonal included in both cases, the lower triangular section of F corresponds to L and its upper triangular section to U. The following is a stylized representation of this matrix:

$$F := \left|\begin{array}{ccccccc|l} a_{1,1}^0 & a_{1,2}^0 & \cdots & a_{1,k}^0 & a_{1,k+1}^0 & \cdots & a_{1,n}^0 & \kappa = 0 \\ a_{2,1}^0 & a_{2,2}^1 & \cdots & a_{2,k}^1 & a_{2,k+1}^1 & \cdots & a_{2,n}^1 & \kappa = 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots \\ a_{k,1}^0 & a_{k,2}^1 & \cdots & a_{k,k}^{k-1} & a_{k,k+1}^{k-1} & \cdots & a_{k,n}^{k-1} & \kappa = k\text{–}1 \\ a_{k+1,1}^0 & a_{k+1,2}^1 & \cdots & a_{k+1,k}^{k-1} & a_{k+1,k+1}^{k} & \cdots & a_{k+1,n}^{k} & \kappa = k \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1}^0 & a_{n,2}^1 & \cdots & a_{n,k}^{k-1} & a_{n,k+1}^{k} & \cdots & a_{n,n}^{n-1} & \kappa = n\text{–}1 \end{array}\right| \tag{6.1}$$

In this representation, F's elements are divided into adjoining *frames* or groups of IPGE entries obtained in the same iteration; the index of each frame is indicated by the right-hand κ-value that aligns with it. Notice that the corner (i.e., diagonal) element of frame k equals the $(k{+}1)$th pivot element used by IPGE, for $1 \leq k \leq n\text{–}1$. Equation (6.1) also indicates that it is only necessary to apply a *reduced version* of

IPGE to obtain REF-LU (or REF-Ch), namely one in which the pivoting operations at iteration k are performed only on the submatrix induced by rows $i > k$ and columns $j > k$ of A^{k-1}. Hence, frame k can be alternatively characterized as the merging of the pivot row and the pivot column utilized in iteration k+1 of the reduced version of IPGE.

The main reason for introducing this notation for REF-LU is to facilitate the subsequent description of the REF factorization update algorithms. For the same reason, the following paragraphs will describe the REF forward and backward substitution algorithms in terms of F even though Section 4 does so in terms of matrices L and U.

Define a triangular array $\Psi = \psi_{i,r}$, for $0 \leq r < i \leq n$, and recall that $\boldsymbol{b}$ is a $n \times 1$ integral vector. Then, the REF forward substitution algorithm is given by the following recursion:

$$\psi_{i,r} = \begin{cases} b_i & \text{if } r = 0 \\ \frac{(f_{r,r}\psi_{i,r-1} - f_{i,r}\psi_{r,r-1})}{f_{r-1,r-1}} & \text{if } 0 < r < i \end{cases} \qquad \text{for } \; i = 1 \ldots n \tag{6.2}$$

where $f_{0,0} = l_{0,0} := 1$. The output of this algorithm is the (REF) $n \times 1$ integral vector $\boldsymbol{y}$, with elements $y_i = \psi_{i,i-1}$, which solves the equation $LD^{-1}\boldsymbol{y} = \boldsymbol{b}$; Lemma 1 in Section 4.3 shows y_i equals IPGE entry $a_{i,n+1}^{i-1}$, where n+1 is the index of the column on which REF forward substitution is performed (i.e., corresponding to $\boldsymbol{b}$). In order to solve $A\boldsymbol{x} = \boldsymbol{b}$ without roundoff errors, however, REF forward and backward substitution must be applied to the scaled SLE $A\boldsymbol{x'} = \boldsymbol{b'}$, where $\boldsymbol{x'} = \det(A)\boldsymbol{x}$ and $\boldsymbol{b'} = \det(A)\boldsymbol{b}$. This scaling guarantees $\boldsymbol{x'}$ is integral during backward substitution—which cannot be guaranteed for $\boldsymbol{x}$—based on Cramer's Rule. Nevertheless, the corresponding vector $\boldsymbol{y'}$ can be equivalently obtained without roundoff error by evaluating

Equation (6.2) and then multiplying the output vector $\boldsymbol{y}$ by $\det(A) = f_{n,n}$. Hence, having calculated $\boldsymbol{y}' = f_{n,n}\boldsymbol{y}$, the REF backward substitution algorithm is given by:

$$x'_i = \frac{1}{f_{i,i}} \left(y'_i - \sum_{j=i+1}^{n} f_{i,j} x'_j \right) \qquad \text{for } \; i = n \ldots 1. \tag{6.3}$$

At the conclusion of this process, the REF solution to the original SLE is thus given by:

$$x_i = \frac{x'_i}{\det(A)} = \frac{x'_i}{f_{n,n}} \qquad \text{for } \; i = 1 \ldots n. \tag{6.4}$$

Therefore, as Expressions (6.2)-(6.4) demonstrate, the REF factorization framework can be equivalently characterized via frame matrix F.

6.3 Traditional and Proposed Methods for Updating REF-LU

Generally speaking, it is a bad idea to apply the traditional delete-insert-reduce approach to update REF-LU. This approach is adapted to the REF factorizations by substituting Gaussian pivots—i.e., the typical row reduction operations—with IPGE pivots. However, the resulting algorithms prove inadequate toward updating REF-LU mainly because they do not consider the existing recursive relationships, or *links*, between the adjacent columns of its L matrix and the adjacent rows of its U matrix or, equivalently, between the adjacent frames of F depicted in Equation (6.1). In particular, frame k encodes pivotal information about frame k–1 that guarantees the exactness of the divisions performed in obtaining the entries of frame k+1 (and the entries of the reduced IPGE iterative matrix A^{k+1}). This dependence is evident from the IPGE formula given by Equation (2.7) in Section 2 (i.e., its traditional definition) or by Equation 6.7 in Section 6.4 (i.e., its generalized definition). Moreover, through the iterative calculation of REF-LU, the information about frame k–1 propagates to

all subsequent frames. Hence, deleting a column of REF-LU and inserting another column in its place significantly disrupts the links between these frames. Section 6.3.1 details the consequences of the frame link disruptions engendered through the traditional insert-delete-reduce approach. Then, Section 6.3.2 outlines the proposed push-and-swap update methodology, which circumvents these disruptions. It accomplishes this by gradually pushing the exiting column out of the basis, updating frame links accordingly, and swapping the exiting and entering columns when doing so affects no frame links.

6.3.1 Inadequacy of the Traditional Delete-Insert-Reduce Update Approach

This subsection focuses on the delete-insert-reduce update approach with respect to column replacement due to the fundamental role this update plays in the application of the LU factorization framework and linear programming as a whole (the column/row addition and deletion adaptations are simpler and are thereby omitted for brevity). In particular, given the LU factorization of basis matrix A, the column replacement update provides a shortcut procedure for obtaining the LU factorization of the adjacent basis matrix $\bar{A}$, which differs from A in only one of its columns and possibly in the order of its rows/columns; this process requires $O(n^2)$ operations or factor-$O(n)$-less operations than the $O(n^3)$ LU factorization process. In this subsection, we define the adjacent basis matrices formally as $A := \boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ and $\bar{A} := P\left(A + (\boldsymbol{a}_{n+1} - \boldsymbol{a}_k)\boldsymbol{e}_k^T\right)Q$; where $\boldsymbol{a}_k$ and $\boldsymbol{a}_{n+1}$ denote the exiting and entering basic columns, respectively; where $\boldsymbol{e}_k$ is the n-length kth elementary vector—i.e., $(\boldsymbol{a}_{n+1} - \boldsymbol{a}_k)\boldsymbol{e}_k^T$ yields an outer product; and, where P and Q are permutation matrices determined at runtime by the update. We denote the corresponding REF factorizations as $LD^{-1}U$ and $\bar{L}\bar{D}^{-1}\bar{U}$.

Traditional update algorithms follow a *delete-insert-reduce* update approach. In

Figure 6.1: Initial steps of the Bartels-Golub update

(a) **Step 0:** Delete $\boldsymbol{a}_k$ and insert $\boldsymbol{y}$ in its place

$$\left[\begin{array}{c|cccc} & y_1 & a^0_{1,k+1} & \cdots & a^0_{1,n} \\ U^{1\ldots k-1}_{1\ldots k-1} & \vdots & \vdots & \ddots & \vdots \\ & y_{k-1} & a^{k-2}_{k-1,k+1} & \cdots & a^{k-2}_{k-1,n} \\ \hline & y_k & a^{k-1}_{k,k+1} & \cdots & a^{k-1}_{k,n} \\ & y_{k+1} & a^{k}_{k+1,k+1} & \cdots & a^{k}_{k+1,n} \\ & \vdots & & \ddots & \vdots \\ & y_m & & & a^{n-1}_{n,n} \end{array}\right]$$

(b) **Step 1:** Shift $\boldsymbol{y}$ to column position n

$$\left[\begin{array}{c|ccc|c} & a^0_{1,k+1} & \cdots & a^0_{1,n} & y_1 \\ U^{1\ldots k-1}_{1\ldots k-1} & \vdots & \ddots & \vdots & \vdots \\ & a^{k-2}_{k-1,k+1} & \cdots & a^{k-2}_{k-1,n} & y_{k-1} \\ \hline & a^{k-1}_{k,k+1} & \cdots & a^{k-1}_{k,n} & y_k \\ & a^{k}_{k+1,k+1} & \cdots & a^{k}_{k+1,n} & y_{k+1} \\ & & \ddots & \vdots & \vdots \\ & & & a^{n-1}_{n,n} & y_n \end{array}\right]$$

particular, they begin by performing forward substitution on $\boldsymbol{a}_{n+1}$ to yield the vector $\boldsymbol{y}$, *deleting* column $U_{(:,k)}$ (i.e., corresponding to the exiting column $\boldsymbol{a}_k$), and *inserting* $\boldsymbol{y}$ in place of the removed column (see Step 0 below). This replacement causes U to lose its upper triangular structure, and existing update algorithms differ in the operations they perform to *reduce* this matrix back to upper triangular form (i.e., to yield $\bar{U}$). In general, the updates reestablish upper triangularity by performing additional Gaussian pivots and by permuting certain rows and columns; the permutations are chosen to minimize the number of computations and/or preserve matrix sparsity. We refer the reader to [26] for a review of popular LU update algorithms.

The inadequacy of employing the delete-insert-reduce update approach on REF-LU is illustrated by modifying the Bartels-Golub algorithm [10] accordingly and examining the consequences. Figure 6.1 depicts Step 0 (described in the preceding paragraph) and Step 1 of this algorithm applied to U (i.e., the upper triangular matrix in REF-LU of basis A).

In Figures 6.1a and 6.1b, blank spaces represent zeros of the matrix and $U^{1\ldots k-1}_{1\ldots k-1}$ is the square submatrix of U that retains its upper triangular structure after the

column replacement procedure (Step 0). Step 1 involves moving $\boldsymbol{y}$ to column position n and shifting columns $U_{(:,k+1)}$ to $U_{(:,n)}$ one position left; the outcome is the upper Hessenberg matrix $\tilde{U}$, which has the advantage of being nearly upper triangular.

Reducing $\tilde{U}$ to upper triangular form requires further pivoting operations on its kth to nth columns. However, performing IPGE pivoting operations to eliminate the sub-diagonal elements of these columns has the potential to be counterintuitively inefficient. This is because, in accordance with the tailored version of IPGE for sparse matrices developed by Lee and Saunders [53], the $(n–k+1) \times (n–k+1)$ upper triangular matrix obtained by eliminating a single nonzero sub-diagonal element via the IPGE pivot (and skipping its trailing zeros) for each column of $\tilde{U}_{k\ldots n}^{k\ldots n}$ is exactly the iterative matrix obtained by performing the reduced version of IPGE on $\tilde{U}_{k\ldots n}^{k\ldots n}$. This means that the determinant of $\tilde{U}_{k\ldots n}^{k\ldots n}$ is inevitably computed in the process and, since the magnitude of an individual entry of $\tilde{U}$ is bounded by $e^{\omega_{\max}}$, the maximum bit-length required by this update is bounded as follows:

$$\lceil \det\left(\tilde{U}_{k\ldots n}^{k\ldots n}\right) \rceil \leq \lceil (n-k+1)\log(e^{\omega_{\max}}\sqrt{n-k+1}) \rceil \tag{6.5}$$

$$= \lceil (n-k+1)(\omega_{\max} + \log\sqrt{n-k+1}) \rceil. \tag{6.6}$$

The above inequality uses Hadamard's bound [43] similar to the derivation of IPGE Property 3. Based on the above expression, therefore, when index k of the exiting column is small, the added complexity created by operand growth alone may be larger than the factor-$O(n)$ operations saved by the Bartels-Golub algorithm (i.e., the added complexity factor is calculated as the cost of multiplying two operands with bit-lengths equal to the above upper bound). The operand growth could be smaller when $\tilde{U}$ is sparse and a specialized update algorithm such as that of Suhl and Suhl [79] is implemented, but not enough to offer a clear advantage computationally over refactorization in many cases.

The delete-insert-reduce update approach encounters two additional significant complications: (1) the output upper triangular matrix, denoted as $\bar{U}'$, does not equal $\bar{U}$ (i.e., the upper triangular matrix obtained by applying the REF LU factorization process directly on basis $\bar{A}$); (2) the process of updating L to $\bar{L}$ is not as straightforward as that of traditional algorithms. The first complication can be inferred from the above operand bit-length analysis, and it is explicitly exhibited via a numerical example in Section 6.6.4. Indeed, $\bar{U}'$ can be thought of as an "inflated" version of $\bar{U}$ and, consequently, it must itself be factored to remove the excess bit-length of its entries. To explain the second complication, assume $\bar{U}$ can actually be obtained from $\bar{U}'$ and that $\bar{L}'$ is the lower triangular matrix induced by the requisite row operations for accomplishing this (i.e., $\bar{L}'\bar{U} = \tilde{U}$). Typically, updating L to $\bar{L}$ involves simply multiplying L by $\bar{L}'$. However, based on the full mathematical expression for REF-LU, the simple variants must work with the right-hand side of the following equation:

$$\bar{L}\bar{D}^{-1} = LD^{-1}\bar{L}'.$$

Hence, the update of L entails first multiplying D^{-1} by $\bar{L}'$; then the non-integral (and non-commutative) product $D^{-1}\bar{L}'$ must be refactored so that $\bar{D}^{-1}$ emerges to the right and the appropriate lower triangular matrix needed to transform L into $\bar{L}$ emerges to the left. The chief difficulty in this protracted process is that these calculations must be done without roundoff error, thereby requiring the use of exact rational arithmetic. This means the advantage of using the REF framework is nullified through the process—i.e., numerators and denominators must be explicitly stored and gcd operations recurrently performed to avoid further growth in their bit-lengths during the matrix multiplication and refactoring process. In short, obtaining $\bar{L}$ in this fashion is not trivial, and it requires considerably more computational effort

than typically expected.

The above analysis attests that the delete-insert-reduce update approach applied to REF-LU does not take into account the special structure of the frame matrix F or, equivalently, of REF-LU's L and U matrices. As a result, its implementation leads to inadequate algorithms. In short, this approach induces a REF but inefficient procedure that recursively updates the rows in the upper triangular part of F (i.e., U), but it does not provide a REF or efficient procedure for updating the columns in the lower triangular part of F (i.e., L).

6.3.2 Outline of the REF Push-and-Swap Update Approach

Piecing together the individual relationships between each adjacent pair of frames of F discussed at the beginning of Section 6.3, it is evident that a modification to frame k–1 (i.e., where column $\boldsymbol{a}_k$ is located) must be complemented by iterative changes to frames k to n–1 (i.e., updates of the affected frame links). In the remainder of this section we develop a REF update methodology that explicitly maintains and updates the links between adjacent frames of F. In each major step of the proposed iterative process, an adjacent pair of frames of the working frame matrix is updated and the recursive dependencies of subsequent frames are maintained and appropriately modified. Thus, the process provides the advantages that the lower and upper triangular parts of F are updated simultaneously and that the final frame matrix corresponds exactly to the REF LU factorization that would be obtained by factoring the target basis matrix $\bar{A}$ directly. Moreover, unlike the aforementioned delete-insert-reduce approach, the individual matrix entries generated in the proposed *push-and-swap approach* require the same maximum bit-length as REF factorization—in other words, they do not cause bit-length growth.

As Section 6.3.1 explains, traditional column replacement algorithms begin by

deleting column $U_{(:,k)}$ and instantly inserting the forward-substituted form of the entering basic column $\boldsymbol{a}_{n+1}$, that is, the vector $\boldsymbol{y}$, in its place. However, the outright removal and replacement of a column in the upper triangular part of F disrupts the links between the column's corresponding and preceding frames in the lower triangular part of F (i.e., L) and, more importantly, its links with subsequent frames. Hence, in order to preserve the special connections between all adjacent frames of F, the proposed methodology instead *pushes* the exiting column gradually to the right until it is safe to *swap* it with the updated form of the entering basic column.

Figure 6.2 depicts the major steps of the push-and-swap update approach with respect to column replacement. The corner element of frame k–1 of F, namely $f_{k,k}$, is by construction the kth pivot element selected by IPGE to solve $A\boldsymbol{x} = \boldsymbol{b}$; in Figure 6.2a, this element is set as $f_{k,k} = a_{k,k}^{k-1}$ according to Assumption 2. Based on this correspondence, Figure 6.2b suggests that an individual push step effectively replaces a given pivot with an adjacent element along the same frame—e.g., in Step 1, $a_{k,k+1}^{k-1}$ replaces $a_{k,k}^{k-1}$ as the corner element of frame k–1. Thus, each push step consists of permuting a frame matrix column, replacing the respective pivot, and relaying the effects of the exchange to the entries in its current and subsequent frames. Once the exiting column is pushed to the rightmost (i.e., nth) basic column position, it is safe to swap it with the modified entering (i.e., nonbasic) column: no frame links are affected at this point and both columns share the same frame structure—i.e., their row entries originate from the same pivots, as Section 6.5 later explains.

The outline of the proposed push-and-swap methodology leads to the following successive implications: (1) performing a series of push steps leads to a specific sequence of pivot choices in IPGE different from the standard pivot sequence fixed by Assumption 2; (2) reflecting the application of these distinct pivot choices on A and of each push step calls for a more general definition of IPGE and the frame matrix

Figure 6.2: Visual outline of the push-and-swap update approach

(a) **Step 0:** Perform REF forward substitution on $\boldsymbol{a}_{n+1}$ to obtain $\boldsymbol{y}$ and append this vector to the right of F (placement demarcated by dotted line); select the index of the exiting column (shaded).

1	$\dots$	k–1	k	k+1	$\dots$	n	n+1	
$a^0_{1,1}$	$\dots$	$a^0_{1,k-1}$	$a^0_{1,k}$	$a^0_{1,k+1}$	$\dots$	$a^0_{1,n}$	y_1	$\kappa = 0$
$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
$a^0_{k-1,1}$	$\dots$	$a^{k-2}_{k-1,k-1}$	$a^{k-2}_{k-1,k}$	$a^{k-2}_{k-1,k+1}$	$\dots$	$a^{k-2}_{k-1,n}$	y_{k-1}	$\kappa = k$–2
$a^0_{k,1}$	$\dots$	$a^{k-2}_{k,k-1}$	$a^{k-1}_{k,k}$	$a^{k-1}_{k,k+1}$	$\dots$	$a^{k-1}_{k,n}$	y_k	$\kappa = k$–1
$a^0_{k+1,1}$	$\dots$	$a^{k-2}_{k+1,k-1}$	$a^{k-1}_{k+1,k}$	$a^{k}_{k+1,k+1}$	$\dots$	$a^{k}_{k+1,n}$	y_{k+1}	$\kappa = k$
$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$a^0_{n,1}$	$\dots$	$a^{k-2}_{n,k-1}$	$a^{k-1}_{n,k}$	$a^{k}_{n,k+1}$	$\dots$	$a^{n-1}_{n,n}$	y_n	$\kappa = n$–1

(b) **Push Step i; for 1≤ i ≤ n–k (i =1 shown):** *Push* the exiting column one position to the right. The elements in frames k+i–2 to n–1 labeled with † are obtained via special REF operations and shortcuts defined in this section. The remaining elements of the matrix are not altered.

1	$\dots$	k–1	k+1	k	$\dots$	n	n+1	
$a^0_{1,1}$	$\dots$	$a^0_{1,k-1}$	$a^0_{1,k+1}$	$a^0_{1,k}$	$\dots$	$a^0_{1,n}$	y_1	$\kappa = 0$
$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
$a^0_{k-1,1}$	$\dots$	$a^{k-2}_{k-1,k-1}$	$a^{k-2}_{k-1,k+1}$	$a^{k-2}_{k-1,k}$	$\dots$	$a^{k-2}_{k-1,n}$	y_{k-1}	$\kappa = k$–2
$a^0_{k,1}$	$\dots$	$a^{k-2}_{k,k-1}$	$a^{k-1}_{k,k+1}$	$a^{k-1}_{k,k}$	$\dots$	$a^{k-1}_{k,n}$	y_k	$\kappa = k$–1
$a^0_{k+1,1}$	$\dots$	$a^{k-2}_{k+1,k-1}$	†	†	$\dots$	†	†	$\kappa = k$
$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$a^0_{n,1}$	$\dots$	$a^{k-2}_{n,k-1}$	†	†	$\dots$	†	†	$\kappa = n$–1

(c) **Swap Step:** *Swap* the exiting column—in the rightmost basic position after completing Step (b)—with the modified entering column—in the nonbasic position. The resulting matrix (shown), minus the shaded column, corresponds to the frame matrix of $\bar{A}$ times a column permutation matrix determined at runtime. The symbols †, ††, and ‡ are used to emphasize that the final form of frames k to n–1 (excluding simple column permutations of their entries) is obtained successively with each push step; the column to the right of the frame matrix indexes the frames.

1	$\dots$	k–1	k+1	k+2	$\dots$	n+1	k	
$a^0_{1,1}$	$\dots$	$a^0_{1,k-1}$	$a^0_{1,k+1}$	$a^0_{1,k+2}$	$\dots$	y_1	$a^0_{1,k}$	$\kappa = 0$
$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
$a^0_{k-1,1}$	$\dots$	$a^{k-2}_{k-1,k-1}$	$a^{k-2}_{k-1,k+1}$	$a^{k-2}_{k-1,k+2}$	$\dots$	y_{k-1}	$a^{k-2}_{k-1,k}$	$\kappa = k$–2
$a^0_{k,1}$	$\dots$	$a^{k-2}_{k,k-1}$	$a^{k-1}_{k,k+1}$	$a^{k-1}_{k,k+2}$	$\dots$	y_k	$a^{k-1}_{k,k}$	$\kappa = k$–1
$a^0_{k+1,1}$	$\dots$	$a^{k-2}_{k+1,k-1}$	†	††	$\dots$	††	††	$\kappa = k$
$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$a^0_{n,1}$	$\dots$	$a^{k-2}_{n,k-1}$	†	††	$\dots$	‡	‡	$\kappa = n$–1

F (i.e., since it was defined only for the standard pivot sequence); (3) efficiently obtaining the exact form of the entries labeled with the symbol † in Figure 6.2b entails developing new (auxiliary) REF operations; (4) replacing a column of REF-LU via the proposed framework depends on first developing other (elementary) updates. The preceding implications provide a road map for the organization of Section 6.4. Then Section 6.5 utilizes this groundwork to develop the featured REF factorization updates.

6.4 Auxiliary and Elementary REF Update Operations

This subsection introduces and proves the correctness of two sets of REF operations. The operations comprising the first set are auxiliary to those in the second set, while those comprising the second set are elementary factorization update operations that function as building blocks for the factorization algorithms developed in Section 6.5.

In essence, the elementary update operations featured in the second part of this subsection provide shortcuts for transitioning between two similar *runs of IPGE on* A, which are required to reflect the output of each push step. An individual run is characterized by a specific n-length sequence of pivots chosen in succession to factor A; the sequence is represented by the corresponding coordinates $(r_1, c_1), \ldots, (r_n, c_n)$ of its elements. Thus, the sequence uniquely determines the specific entries of the REF L and U matrices and, by extension, of the corresponding frame matrix. To enable the representation of multiple runs of IPGE on A, *Assumption 2 is henceforth removed*. Furthermore, for the same purpose, the ensuing paragraphs broaden the definition of IPGE, its general connection to specific subdeterminants of A (i.e., IPGE Property 2), and the definition of a frame matrix.

Definition 10. *A feasible pivot sequence is represented via an array* $\varrho[\cdot]$ *of ordered*

pairs with individual elements $\varrho[1] := (r_1, c_1), \varrho[2] := (r_2, c_2), \ldots, \varrho[n] := (r_n, c_n)$ *such that each row coordinate* r_k *and each column coordinate* c_k *is unique and such that the* k*th pivot,* ρ^k*, referenced by array element* $\varrho[k] = (r_k, c_k)$*, is nonzero. By default, the singleton* $\varrho[0] := 0$ *is appended to every feasible sequence of pivot coordinates (recall* $\rho^0 = a_{0,0}^{-1} = 1$*).*

Definition 11. *Let* $a_{i,j}^{\varrho[k]}$*, or equivalently* $a_{i,j}^{(r_k,c_k)}$*, denote the* k*th-iteration* (i,j)*-entry that IPGE obtains by using pivot sequence* ϱ*, where* $1 \leq i, j \leq n$ *and* $0 \leq k \leq n$*. In particular, this notation indicates the element with coordinates* $\varrho[k] = (r_k, c_k)$ *in the corresponding* $(k$–$1)$*th IPGE iterative matrix,* $A^{\varrho[k-1]}$*, is selected as the* k*th pivot element,* ρ^k*. As a convention,* $a_{i,j}^{\varrho[0]} := a_{i,j}^0 = a_{i,j}$*.*

Connecting each IPGE entry with the coordinates of the pivot from which it was obtained—in other words, with its *originating pivot*—leads to a generalized description of the IPGE algorithm (the traditional version is given by Equation (2.7)):

$$a_{i,j}^{\varrho[k]} = \begin{cases} a_{i,j}^{\varrho[k-1]} & \text{if } i = r_k \\ \left(a_{\varrho[k]}^{\varrho[k-1]} a_{i,j}^{\varrho[k-1]} - a_{r_k,j}^{\varrho[k-1]} a_{i,c_k}^{\varrho[k-1]} \right) / a_{\varrho[k-1]}^{\varrho[k-2]} & \text{otherwise} \end{cases} \quad \text{for } \; k = 1 \ldots n \quad (6.7)$$

where $a_{\varrho[k]}^{\varrho[k-1]}$ denotes the succinct representation of the kth pivot, $a_{r_k,c_k}^{\varrho[k-1]}$, via a natural extension of this notation. A benefit of the generalized definition of IPGE given by Equation (6.7) is that the candidate pivot rows and columns remaining at step k of IPGE are easily deduced by eliminating from contention the row and column coordinates stored in $\varrho[1], \ldots, \varrho[k$–$1]$ successively. Consequently, it is unnecessary to keep track of iterative permutation matrices. The relationship between $a_{i,j}^k$ and the

subdeterminants of A (see Equation (2.11)) is generalized to $a_{i,j}^{\varrho[k]}$ as follows:

$$a_{i,j}^{\varrho[k]} = \begin{cases} (-1)^{\kappa(i)+k} \det\left((A)_{\{c_1 \ldots c_k\} \setminus c_{\kappa(i)}, j}^{r_1 \ldots r_k}\right) & \text{if } i \in \{r_1, \ldots, r_k\} \\ \det\left((A)_{c_1 \ldots c_k, j}^{r_1 \ldots r_k, i}\right) & \text{if } i \in \{r_{k+1}, \ldots, r_n\} \end{cases} \tag{6.8}$$

where $(A)_{c_1 \ldots c_k, j}^{r_1 \ldots r_k, i}$ is the submatrix induced by rows r_1 to r_k and i and columns c_1 to c_k and j of A; and where $\kappa(i)$ is the index such that $r_\kappa = i$ (i.e., the IPGE iteration index in which row i was used as a pivot row).

Section 6.2 explained that the frames of F are exactly the pivot rows and the pivot columns used by the (reduced version) IPGE run on A associated with the standard pivot sequence given by Assumption 2. Extending this connection, the next definition generalizes F to the frame matrix obtained via the IPGE run on A associated with ϱ, denoted as F^{ϱ}.

Definition 12. *The individual entries of frame matrix $F^{\varrho} : (A, \varrho) \to \mathbb{Z}^{n \times n}$, obtained by factoring A according to feasible pivot sequence ϱ, are given by:*

$$f_{i,j}^{\varrho} := a_{r_i, c_j}^{\varrho[\underline{k}]} \tag{6.9}$$

where $\underline{k} = \min(i, j) - 1$ and $1 \leq i, j \leq n$. Thus, combining Equations (6.8) and (6.9) gives:

$$f_{i,j}^{\varrho} = \det\left((A)_{c_1 \ldots c_{\underline{k}}, c_j}^{r_1 \ldots r_{\underline{k}}, r_i}\right) \tag{6.10}$$

F^{ϱ} has equivalent properties as F, except that it corresponds to the REF LU factorization of A obtained using feasible pivot sequence ϱ instead of the standard sequence induced by Assumption 2. Accordingly, it fully characterizes the REF fac-

torization SLE-solution framework (see Expressions (6.2), (6.3), and (6.4)) associated with the ordered basis induced by rows $r_1, \ldots, r_n$ and columns $c_1, \ldots, c_n$ of A.

6.4.1 Auxiliary REF Operations

This subsection develops two new REF operations: backtracking and row-wise switch of originating pivot (RwSOP). In essence, these operations provide shortcuts for backtracking the iteration counter of an IPGE entry—i.e., for obtaining the value the entry had in the previous iteration—and for replacing its originating pivot with an element from the same pivot row, respectively. We direct the reader to Figures 6.7b and 6.7c in Section 6.6 to see an application of each auxiliary operation on a numerical example.

Theorem 6.4.1 (Backtracking)**.** *For $k > 0$, the $(k$–$1)$-iteration entry $a_{i,j}^{\varrho[k-1]}$ can be obtained* **without roundoff error** *from its corresponding k-iterate, $a_{i,j}^{\varrho[k]}$, as follows:*

$$a_{i,j}^{\varrho[k-1]} = \begin{cases} a_{i,j}^{\varrho[k]} & \text{if } i = r_k \\ \left(a_{\varrho[k-1]}^{\varrho[k-2]} a_{i,j}^{\varrho[k]} + a_{r_k,j}^{\varrho[k-1]} a_{i,c_k}^{\varrho[k-1]} \right) / a_{\varrho[k]}^{\varrho[k-1]} & \text{otherwise} \end{cases} \tag{6.11}$$

Proof. When $i = r_k$, the result holds trivially from Equation (6.7). When $i \neq r_k$, multiplying Equation (6.7) by $a_{\varrho[k-1]}^{\varrho[k-2]}$ gives:

$$a_{\varrho[k-1]}^{\varrho[k-2]} a_{i,j}^{\varrho[k]} = a_{\varrho[k]}^{\varrho[k-1]} a_{i,j}^{\varrho[k-1]} - a_{r_k,j}^{\varrho[k-1]} a_{i,c_k}^{\varrho[k-1]},$$

and substituting this expression into the right-hand side of Equation (6.11) gives:

$$\frac{\left(a_{\varrho[k]}^{\varrho[k-1]} a_{i,j}^{\varrho[k-1]} - a_{r_k,j}^{\varrho[k-1]} a_{i,c_k}^{\varrho[k-1]} \right) + a_{r_k,j}^{\varrho[k-1]} a_{i,c_k}^{\varrho[k-1]}}{a_{\varrho[k]}^{\varrho[k-1]}} = \frac{a_{\varrho[k]}^{\varrho[k-1]} a_{i,j}^{\varrho[k-1]}}{a_{\varrho[k]}^{\varrho[k-1]}} = a_{i,j}^{\varrho[k-1]}.$$

Notice that the numerator in the left-hand side of the second equation is an exact

multiple of its denominator and, therefore, the formula is REF. □

Continuing with the featured terminology, let ϱ and ϱ' represent two runs of IPGE on A with identical pivot selections in iterations 1 to k–1 but with differing selections in iteration k. Formally, $\varrho[h] = \varrho'[h]$ for $1 \leq h < k$ (i.e., $r'_h = r_h$ and $c'_h = c_h$), but $\varrho[k] \neq \varrho'[k]$ (i.e., $r'_h \neq r_h$ and/or $c'_h \neq c_h$). Hence, at the conclusion of k iterations, entries $a_{i,j}^{\varrho[k]}$ and $a_{i,j}^{\varrho'[k]}$ were obtained using different originating pivots ρ^k and $(\rho')^k$ (selected from identical (k–1)-iteration IPGE iterative matrices $A^{\varrho[k-1]}$ and $A^{\varrho'[k-1]}$). We note that, when performing only the IPGE run associated with ϱ, if $A^{\varrho[k-1]}$ is overwritten with $A^{\varrho[k]}$ during iteration k, it is still possible to obtain $a_{i,j}^{\varrho'[k]}$. This can be done by backtracking the entries $a_{r'_k,c'_k}^{\varrho[k]}$, $a_{i,j}^{\varrho[k]}$, $a_{r'_k,j}^{\varrho[k]}$, and $a_{i,c'_k}^{\varrho[k]}$ once and the entry $a_{\varrho[k-1]}^{\varrho[k]} = a_{\varrho'[k-1]}^{\varrho[k]}$ twice, and then solving the right-hand side of Equation (6.7) with respect to $a_{i,j}^{\varrho'[k]}$. As the following theorem demonstrates, however, when $(\rho')^k$ is chosen from the same row as ρ^k, a shortcut can be applied that circumvents this process. Afterward, the subsequent paragraph explains the shortcut's relevance to F^{ϱ}.

Theorem 6.4.2 (Row-wise Switch of Origination Pivot—RwSOP)**.** *Let ϱ' be a feasible pivot sequence with identical pivot selections as ϱ in iterations 1 to k–1 but with a kth pivot selection from the same row but from a different column than ρ^k. Then, for $i \neq r'_k = r_k$, IPGE entry $a_{i,j}^{\varrho'[k]}$ can be equivalently obtained* **without roundoff error** *via the formula:*

$$a_{i,j}^{\varrho'[k]} = \frac{a_{\varrho'[k]}^{\varrho[k-1]} a_{i,j}^{\varrho[k]} - a_{r'_k,j}^{\varrho[k-1]} a_{i,c'_k}^{\varrho[k]}}{a_{\varrho[k]}^{\varrho[k-1]}} \tag{6.12}$$

Proof. Substituting elements $a_{i,j}^{\varrho[k]}$ and $a_{i,c'_k}^{\varrho[k]}$ with their corresponding definition given

by Equation (6.7), the right-hand side of the above equation becomes:

$$\frac{a_{\varrho'[k]}^{\varrho[k-1]}\left(a_{\varrho[k]}^{\varrho[k-1]}a_{i,j}^{\varrho[k-1]}-a_{r_k,j}^{\varrho[k-1]}a_{i,c_k}^{\varrho[k-1]}\right)-a_{r'_k,j}^{\varrho[k-1]}\left(a_{\varrho[k]}^{\varrho[k-1]}a_{i,c'_k}^{\varrho[k-1]}-a_{r_k,c'_k}^{\varrho[k-1]}a_{i,c_k}^{\varrho[k-1]}\right)}{a_{\varrho[k]}^{\varrho[k-1]}a_{\varrho[k-1]}^{\varrho[k-2]}} \tag{6.13}$$

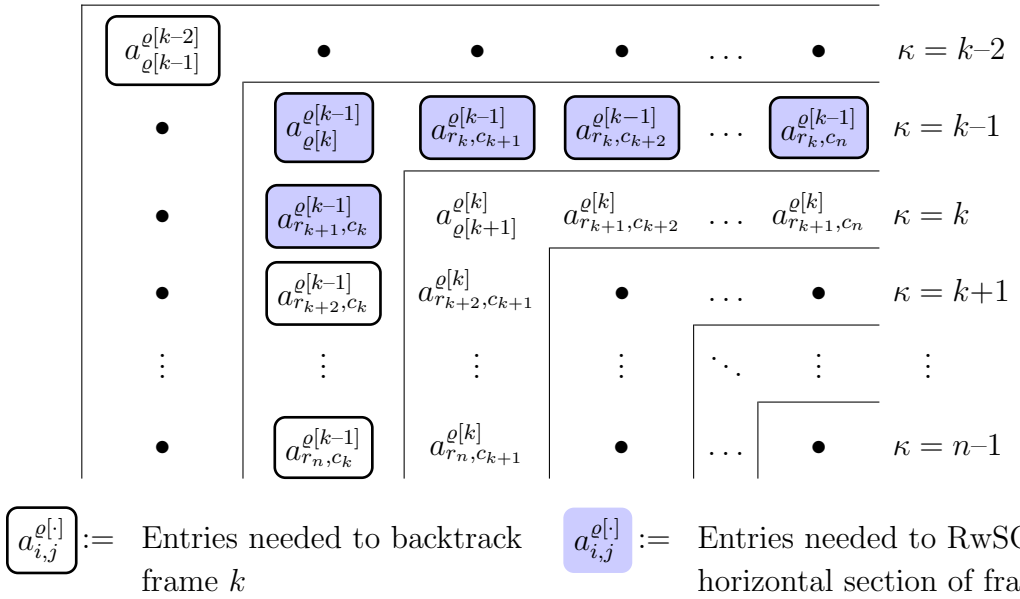
$$=\frac{a_{\varrho'[k]}^{\varrho[k-1]}\left(a_{\varrho[k]}^{\varrho[k-1]}a_{i,j}^{\varrho[k-1]}-\underline{a_{r'_k,j}^{\varrho[k-1]}}a_{i,c_k}^{\varrho[k-1]}\right)-a_{r'_k,j}^{\varrho[k-1]}\left(a_{\varrho[k]}^{\varrho[k-1]}a_{i,c'_k}^{\varrho[k-1]}-\underline{a_{\varrho'[k]}^{\varrho[k-1]}}a_{i,c_k}^{\varrho[k-1]}\right)}{a_{\varrho[k]}^{\varrho[k-1]}a_{\varrho[k-1]}^{\varrho[k-2]}} \tag{6.14}$$

$$=\frac{\cancel{a_{\varrho[k]}^{\varrho[k-1]}}\left(a_{\varrho'[k]}^{\varrho[k-1]}a_{i,j}^{\varrho[k-1]}-a_{r'_k,j}^{\varrho[k-1]}a_{i,c'_k}^{\varrho[k-1]}\right)}{\cancel{a_{\varrho[k]}^{\varrho[k-1]}}a_{\varrho[k-1]}^{\varrho[k-2]}}=a_{i,j}^{\varrho'[k]} \tag{6.15}$$

where Equation (6.14) substitutes r_k with r'_k (since $r'_k = r_k$) and then $a_{r'_k,c'_k}^{\varrho[k-1]}$ (not shown) with $a_{\varrho'[k]}^{\varrho[k-1]}$ by the equivalence of both notations (the substituted entries are underlined); where the first equation in Expression (6.15) cancels the second and fourth numerator term of Equation (6.14) and regroups terms; and where the second equation cancels the common factor $a_{\varrho[k]}^{\varrho[k-1]}$ and applies the definition of $a_{i,j}^{\varrho'[k]}$. The above derivation involved two substitutions, an exact cancelation, and an exact division; thus, the formula is REF. □

Before proceeding, it is important to point out that the ensuing elementary update operations will apply an auxiliary REF operation to an entire frame or to a large section of it. As Equation (6.11) indicates and the partial frame matrix in Figure 6.3 illustrates, frames k–1 and k–2 contain all the entries needed to backtrack all of frame k. Moreover, based on Equation (6.12), frame k–1 contains all the entries needed to perform a row-wise switch of originating pivot (RwSOP) on the horizontal section of frame k—i.e., entries $f_{k+1,k+1}^{\varrho}$ to $f_{k+1,n}^{\varrho}$. (The visual depiction omits frames 0 to k–3 and uses large bullets to emphasize that the omitted entries are inactive during these operations.)

Figure 6.3: Matrix entries needed to backtrack a frame and to RwSOP the horizontal section of a frame



$a^{\varrho[\cdot]}_{i,j}$:= Entries needed to backtrack frame k

$a^{\varrho[\cdot]}_{i,j}$:= Entries needed to RwSOP the horizontal section of frame k

6.4.2 Elementary Update Operations

The elementary update operations combine auxiliary and traditional IPGE operations, as well as other shortcuts, to perform the individual push steps involved in the featured REF factorization update process. For the remainder of this section, let A represent the initial nonsingular matrix and $\bar{A}$ its updated counterpart. Similarly, let F^{ϱ} and $\bar{F}^{\varrho}$ denote the frame matrices of A and $\bar{A}$, respectively, obtained via a pivot sequence ϱ that is feasible for A. The discussion of each update specifies the conditions for ϱ to be feasible for $\bar{A}$ as well as the structure of A and $\bar{A}$. The objective of each ensuing update discussion, including those of Section 6.5, is to describe efficient procedures for obtaining $\bar{F}^{\varrho}$ given F^{ϱ}.

In the ensuing elementary updates, $\bar{A}$ is obtained from A via a row and/or column permutation. Hence, it is possible to apply a pivot sequence ϱ' to A that results in the

same pivot choices as ϱ applied to $\bar{A}$ by permuting the row and/or column coordinates of ϱ according to the permutations needed to obtain $\bar{A}$ from A. Moreover, since $\bar{A}$'s rows and columns are arranged in $\bar{F}^{\varrho}$ according to the order its pivots are selected, $\bar{F}^{\varrho}$ is equivalent to $F^{\varrho'}$, which represents the IPGE run on A associated with ϱ'.

6.4.2.1 Adjacent Pivot Column Permutation (i.e., Column Push).

Let $A_{(:,c_k)}$ denote column vector c_k of A, and let $\boldsymbol{e}_{c_k}$ denote the n-length (c_k)th elementary vector. Then, the equation,

$$\bar{A} := A + (A_{(:,c_{k+1})} - A_{(:,c_k)})\boldsymbol{e}_{c_k}^T + (A_{(:,c_k)} - A_{(:,c_{k+1})})\boldsymbol{e}_{c_{k+1}}^T \tag{6.16}$$

defines the matrix obtained by interchanging columns c_k and c_{k+1} of A. Define ϱ' as the pivot sequence that swaps only column indices k and k+1 of ϱ with respect to A; that is, ϱ' differs from ϱ only in that $c'_k = c_{k+1}$ and $c'_{k+1} = c_k$. This subsection describes how to obtain $F^{\varrho'}$ (i.e., $\bar{F}^{\varrho}$) given F^{ϱ} efficiently.

The derivation of the Adjacent Pivot Column Permutation Update algorithm (APCPU) is organized as follows. First, Lemma 3 establishes that all that is required for ϱ' to be feasible for A is $f^{\varrho}_{k,k+1} \neq 0$. This result is not trivial because it guarantees that the $(k$+2)th to nth pivots referenced by ϱ'—which have the same row and column coordinates as the $(k$+2)th to nth pivots referenced by ϱ—are nonzero even though their values recursively change when c_{k+1} is chosen instead of c_k as the kth-iteration pivot column. Second, Theorem 6.4.3 provides shortcuts for APCPU by demonstrating that most of the updated entries in the transition from F^{ϱ} to $F^{\varrho'}$ can be obtained via simple sign changes. Third, Figure 6.4 offers a step-by-step description and accompanying visual representation of APCPU; we direct the reader to Figure 6.7 in Section 6.6 to see the application of this update on a numerical example. Fourth, Theorem 6.4.4 proves that the algorithm is correct and REF.

Fifth, Corollary 5 derives its number of operations.

Lemma 3. *Let ϱ' be the pivot sequence obtained from swapping column indices k and $k+1$ of a pivot sequence ϱ that is feasible for A. Then, ϱ' is feasible for A if and only if $f^{\varrho}_{k,k+1} \neq 0$.*

Proof. The uniqueness of each row and column coordinate of ϱ' is proved from combining the fact ϱ is a feasible pivot sequence with the observations $\left\{ \bigcup_{k=1}^{n} r'_k \right\} = \left\{ \bigcup_{k=1}^{n} r_k \right\}$ and $\left\{ \bigcup_{k=1}^{n} c'_k \right\} = \left\{ \bigcup_{k=1}^{n} c_k \right\}$ (i.e., the row and column coordinate sets each have the same n elements). To determine if all the elements referenced by ϱ' are nonzero, first notice that, for $0 \leq h \leq k{-}1$, $(\rho')^h = \rho^h \neq 0$ because ϱ is a feasible pivot sequence. However, $(\rho')^k$ is nonzero if and only if $a^{\varrho[k-1]}_{r_k,c_{k+1}} = f^{\varrho}_{k,k+1} \neq 0$. Assuming $(\rho')^k \neq 0$, for $h = k{+}1 \ldots n$, the following series of equalities can be established regarding the new hth pivot, $a^{\varrho'[h-1]}_{\varrho'[h]}$:

$$a^{\varrho'[h-1]}_{\varrho'[h]} = \det\left((A)^{r'_1 \ldots r'_h}_{c'_1 \ldots c'_h}\right) = \quad \det\left((A)^{r_1 \ldots r_h}_{c_1 \ldots c_{k-1}, c_{k+1}, c_k, c_{k+2} \ldots c_h}\right) \tag{6.17}$$

$$= -\det\left((A)^{r_1 \ldots r_h}_{c_1 \ldots c_{k-1}, c_k, c_{k+1}, c_{k+2} \ldots c_h}\right) = -a^{\varrho[h-1]}_{\varrho[h]} \tag{6.18}$$

where the first equality of Expression (6.17) results from the subdeterminant identities given by Equation (6.8) and the second from the definition of ϱ'; and where the first equality of Expression (6.18) applies the fact that switching two columns of a matrix changes only the sign of its determinant, and the second applies the subdeterminant identities given by Equation (6.8). Since $a^{\varrho[h-1]}_{\varrho[h]} \neq 0$, the preceding expressions imply $(\rho')^h \neq 0$ for $k{+}1 \leq h \leq n$ and, therefore, ϱ' is a feasible pivot sequence for A if and only if $f^{\varrho}_{k,k+1} \neq 0$. □

Theorem 6.4.3 (APCPU Shortcuts)**.** *For $i, j \geq k{+}2$ (i.e., frames $k{+}1$ to $n{-}1$ of*

$F^{\varrho'}$*),*

$$f_{i,j}^{\varrho'} = -f_{i,j}^{\varrho}; \tag{6.19}$$

in addition, for the $(k{+}1)$*th-column entries with row index* $i \geq k{+}1$ *(i.e., the vertical segment of frame* k*),*

$$f_{i,k+1}^{\varrho'} = -f_{i,k+1}^{\varrho}. \tag{6.20}$$

Proof. Let $\underline{k} = \min(i,j) - 1$. Combining the definition of ϱ' and the subdeterminant identity of $f_{i,j}^{\varrho'}$ given by Equation (6.10) gives that, for $i, j \geq k{+}2$,

$$f_{i,j}^{\varrho'} = \det\left((A)_{c_1 \ldots c_{k-1}, c_{k+1}, c_k, c_{k+2} \ldots c_{\underline{k}}, c_j}^{r_1 \ldots r_{\underline{k}}, r_i}\right) = -\det\left((A)_{c_1 \ldots c_{\underline{k}}, c_j}^{r_1 \ldots r_{\underline{k}}, r_i}\right) = -f_{i,j}^{\varrho};$$

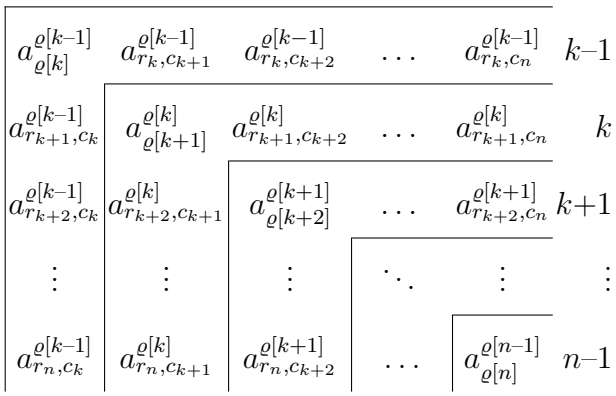
where the second equation applies the fact that swapping two columns of a matrix changes only the sign of its determinant, and the third equation applies the subdeterminant identity of $f_{i,j}^{\varrho}$. Similarly, for the $(k{+}1)$-column entries with row index $i \geq k{+}1$,

$$f_{i,k+1}^{\varrho'} = \det\left((A)_{c_1 \ldots c_{k-1}, c_{k+1}, c_k}^{r_1 \ldots r_k, r_i}\right) = -\det\left((A)_{c_1 \ldots c_{k+1}}^{r_1 \ldots r_k, r_i}\right) = -f_{i,k+1}^{\varrho}. \ \square$$

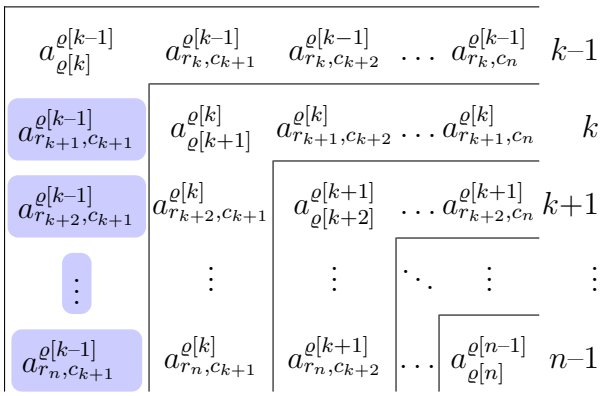
Except for a column permutation, the entries in frames 0 to k–2 are inactive during the APCPU algorithm. For the purpose of clarity, therefore, the algorithm description given in Figure 6.4 omits these frames. Moreover, within each subfigure, blue boxes surround either the individual entries that change in the corresponding APCPU step or the frame index (now stated without using κ) when the full frame changes.

Figure 6.4: The Adjacent Pivot Column Permutation Update algorithm

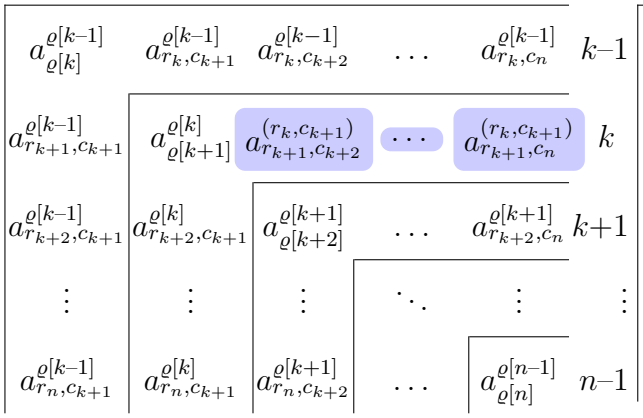
(a) **Input:** F^{ϱ}, k. **Note:** the update $F^{\varrho'}$ is feasible if and only if $f^{\varrho}_{k,k+1} = a^{\varrho[k-1]}_{r_k,c_{k+1}} \neq 0$



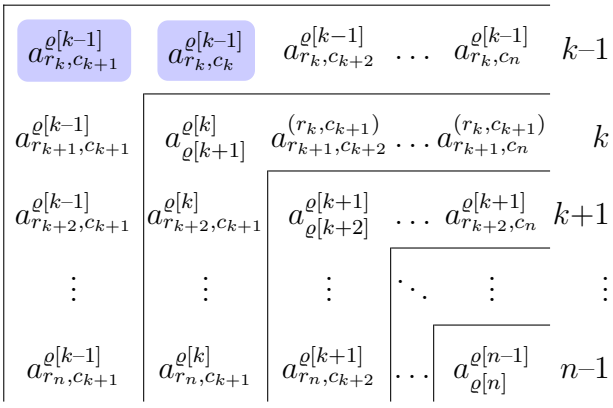
(b) **Step 1:** Backtrack the entries in the vertical portion of frame k and place these entries in the corresponding rows of frame k–1.



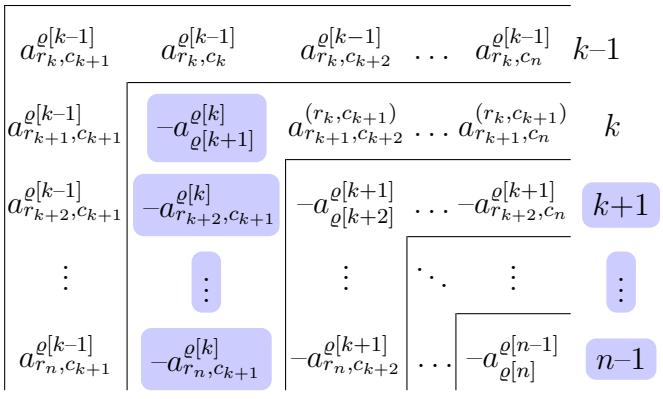
(c) **Step 2:** Perform row-wise switches of originating pivot on the entries in the strictly horizontal segment of frame k using $a^{\varrho[k-1]}_{r_k,c_{k+1}}$ as the new pivot.



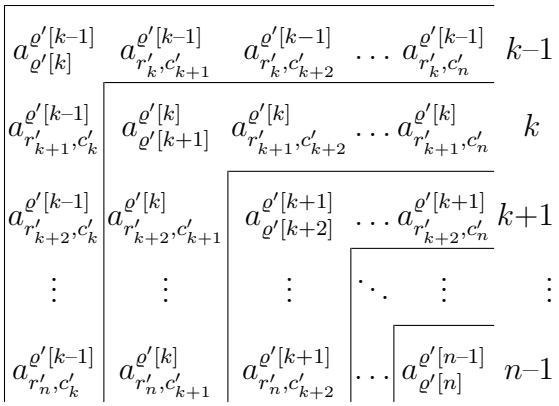
(d) **Step 3:** Permute columns k and $k{+}1$ of frames 0 to k–1 (effect on frames 0 to k–2 not shown).



(e) **Step 4:** Flip the signs of the entries specified by Theorem 6.4.3.



(f) **Output:** $F^{\varrho'}$.



Theorem 6.4.4. *The APCPU algorithm is correct and REF.*

Proof. Let $i \geq k{+}1$. By Equation (6.11), $f^{\varrho}_{i,k+1} = a^{\varrho[k]}_{r_i,c_{k+1}}$ is backtracked via the formula:

$$a^{\varrho[k-1]}_{r_i,c_{k+1}} = \frac{a^{\varrho[k-2]}_{\varrho[k-1]} a^{\varrho[k]}_{r_i,c_{k+1}} + a^{\varrho[k-1]}_{r_k,c_{k+1}} a^{\varrho[k-1]}_{r_i,c_k}}{a^{\varrho[k-1]}_{\varrho[k]}} = \frac{f^{\varrho}_{k-1,k-1} f^{\varrho}_{i,k+1} + f^{\varrho}_{k,k+1} f^{\varrho}_{k+1,k}}{f^{\varrho}_{k,k}}, \tag{6.21}$$

which is REF by Theorem 6.4.1. By definition of ϱ', $a^{\varrho[k-1]}_{r_i,c_{k+1}}$ is equivalent to $a^{\varrho'[k-1]}_{r_i,c'_k} = f^{\varrho'}_{i,k}$. More generally, $F^{\varrho'}$'s entries in frames 0 to k–2 match those of F^{ϱ}, except that column indices c_k and c_{k+1} are reversed, since $\varrho'[h] = \varrho[h]$ for $h \neq k, k{+}1$; the same holds true for those along the horizontal segment of frame k–1 since they are all connected to the originating pivot with coordinates $\varrho'[k{-}1] = \varrho[k{-}1]$. Hence, placing the backtracked entries from the vertical part of frame k into column k (i.e., Step 1) and permuting columns k and $k{+}1$ for frames 0 to k–1 (i.e., Step 3) make the 0 to k–1 modified frames equal to frames 0 to k–1 of $F^{\varrho'}$.

Now, let $j \geq k{+}2$. Applying the RwSOP operation, which is REF by Theorem 6.4.2, the entry $f^{\varrho}_{k+1,j} = a^{\varrho[k]}_{r_{k+1},c_j}$ changes its originating pivot to $(\rho')^k = f^{\varrho}_{k,k+1} = a^{\varrho[k-1]}_{r_k,c'_{k+1}}$ via the formula:

$$a^{\varrho'[k]}_{r_{k+1},c_j} = \frac{a^{\varrho[k-1]}_{\varrho'[k]} a^{\varrho[k]}_{r_{k+1},c_j} - a^{\varrho[k-1]}_{r'_k,c_j} a^{\varrho[k]}_{r_{k+1},c'_k}}{a^{\varrho[k-1]}_{\varrho[k]}} = \frac{f^{\varrho}_{k,k+1} f^{\varrho}_{k+1,j} - f^{\varrho}_{k,j} f^{\varrho}_{k+1,k+1}}{f^{\varrho}_{k,k}}.$$

By definition of $F^{\varrho'}$, the updated entry equals $a^{\varrho'[k]}_{r_{k+1},c_j} = f^{\varrho'}_{k+1,j}$. Flipping the signs of the entries in frames $k{+}1$ to n–1 and in the vertical segment of frame k according to Theorem 6.4.3, therefore, makes the k to n–1 modified frames equal to frames k to n–1 of $F^{\varrho'}$. Thus, APCPU is correct and REF since its constituent operations are REF.□

Prior to deriving the number of operations APCPU requires, we specify an important assumption regarding the implementation of sign flips on full frames of a frame matrix.

Assumption 3. *When implementing any of the REF factorization update algorithms, flipping the signs of all the entries in a given frame is not done explicitly. Instead, using an n-length array, $\boldsymbol{s}$, storing possible values ± 1, the sign changes of each corresponding frame are stored as follows: when the signs of all the entries of frame k are flipped, $s[k]$ is set to -1 if $s[k] = 1$, or vice versa, for $0 \leq k \leq n-1$. Then, to represent its sign at any particular juncture, frame entry $f^{\varrho}_{i,j}$ actually has the extended form $s[\underline{k}]f^{\varrho}_{i,j}$, for $1 \leq i, j \leq n$, where $\underline{k} = \min(i, j) - 1$. Based on this implementation, therefore, we claim that flipping the signs of $O(n)$ frames requires $O(n)$ operations. Moreover, we assume that when any of the REF factorization updates featured in Section 6.5 concludes, the extended expression of every entry of the updated frame matrix is evaluated in order to store each product as a single integer; this process requires $O(n^2)$ operations. For the purpose of clarity, however, the descriptions of each algorithm omit these implementation details.*

Corollary 5. *APCPU requires $O(n)$ operations.*

Proof. A backtracking/RwSOP operation consists of two multiplications, one division, and one addition. Hence, backtracking the vertical part of frame k and applying RwSOP to its horizontal part requires $O(n)$ operations. Similarly, permuting columns k and $k+1$ and flipping the signs of the entries in frames k to $n-1$ requires $O(n)$ operations. □

6.4.2.2 Adjacent Pivot Diagonal Permutation (i.e., Column and Row Push).

Let $A_{(r_k:)}$ denote row vector r_k of A. Then, the equation,

$$\begin{aligned}\bar{A} := A &+ (A_{(:,c_{k+1})} - A_{(:,c_k)})\boldsymbol{e}_{c_k}^T + (A_{(:,c_k)} - A_{(:,c_{k+1})})\boldsymbol{e}_{c_{k+1}}^T \\ &+\boldsymbol{e}_{r_k}(A_{(r_{k+1},:)} - A_{(r_k,:)}) + \boldsymbol{e}_{r_{k+1}}(A_{(r_k,:)} - A_{(r_{k+1},:)})\end{aligned}$$

defines the matrix obtained by interchanging columns c_k and c_{k+1} and rows r_k and r_{k+1} of A. Define ϱ' as the pivot sequence that swaps row and column indices k and k+1 of ϱ with respect to A; that is ϱ' differs from ϱ only in that $\varrho'[k] = \varrho[k+1]$ and $\varrho'[k+1] = \varrho[k]$. This subsection describes how to obtain $F^{\varrho'}$ (i.e., $\bar{F}^{\varrho}$) given F^{ϱ}. The derivation of the Adjacent Pivot Diagonal Permutation Update algorithm (APDPU) follows a similar blueprint as that of the APCPU algorithm and, thus, we strive for brevity whenever possible.

Lemma 4. *Let ϱ' be the pivot sequence obtained from swapping row and column indices k and k+1 of a pivot sequence ϱ that is feasible for A. Then, ϱ' is feasible for A if and only if $f^{\varrho}_{k-1,k-1}f^{\varrho}_{k+1,k+1} \neq -f^{\varrho}_{k,k+1}f^{\varrho}_{k+1,k}$.*

Proof. Since $\left\{\bigcup_{k=1}^{n} r'_k\right\} = \left\{\bigcup_{k=1}^{n} r_k\right\}$ and $\left\{\bigcup_{k=1}^{n} c'_k\right\} = \left\{\bigcup_{k=1}^{n} c_k\right\}$, the row and column coordinates in ϱ' are unique. From the feasibility of ϱ and the definition of ϱ', $(\rho')^h = \rho^h \neq 0$ for $0 \leq h \leq k$–1. For showing the feasibility of pivots k to n–1 in ϱ', recall $f^{\varrho}_{k+1,k+1} = a^{\varrho[k]}_{r_{k+1},c_{k+1}}$ by definition of F^{ϱ} and, by Theorem 6.4.1, the backtracked version of this entry is given by,

$$a^{\varrho[k-1]}_{r_{k+1},c_{k+1}} = \frac{a^{\varrho[k-2]}_{\varrho[k-1]}a^{\varrho[k]}_{r_{k+1},c_{k+1}} + a^{\varrho[k-1]}_{r_k,c_{k+1}}a^{\varrho[k-1]}_{r_{k+1},c_k}}{a^{\varrho[k-1]}_{\varrho[k]}} = \frac{f^{\varrho}_{k-1,k-1}f^{\varrho}_{k+1,k+1} + f^{\varrho}_{k,k+1}f^{\varrho}_{k+1,k}}{f^{\varrho}_{k,k}}.$$

This implies $a^{\varrho[k-1]}_{r_{k+1},c_{k+1}}$ is nonzero if and only if $f^{\varrho}_{k-1,k-1}f^{\varrho}_{k+1,k+1} \neq -f^{\varrho}_{k,k+1}f^{\varrho}_{k+1,k}$ since

the first numerator term is a product of two pivot elements, which are nonzero by definition. Consequently, since $a^{\varrho[k-1]}_{r_{k+1},c_{k+1}} = a^{\varrho'[k-1]}_{r'_k,c'_k}$ by definition of ϱ', meeting this condition guarantees $(\rho')^k$ is nonzero. Moreover, assuming $(\rho')^k \neq 0$, for $k < h < n$, from Equation (6.8) and the definition of ϱ',

$$a^{\varrho'[h-1]}_{\varrho'[h]} = \det\left((A)^{r_1 \ldots r_{k-1}, r_{k+1}, r_k, r_{k+2} \ldots r_h}_{c_1 \ldots c_{k-1}, c_{k+1}, c_k, c_{k+2} \ldots c_h}\right) = \det\left((A)^{r_1 \ldots r_h}_{c_1 \ldots c_h}\right) = a^{\varrho[h-1]}_{\varrho[h]}$$

owing to the fact that swapping two rows and two columns of a matrix does not change its determinant. Since $a^{\varrho[h-1]}_{h,h} \neq 0$, the preceding expression implies $(\rho')^h \neq 0$ for $k{+}1 \leq h \leq n$ and, therefore, ϱ' is a feasible pivot sequence. □

Theorem 6.4.5. *For $i, j \geq k{+}2$ (i.e., frames $k{+}1$ to n of $F^{\varrho'}$), $f^{\varrho'}_{i,j} = f^{\varrho}_{i,j}$.*

Proof. Let $\underline{k} = \min(i, j) - 1$. For $i, j \geq k{+}2$, the subdeterminant identity of $f^{\varrho}_{i,j}$ gives,

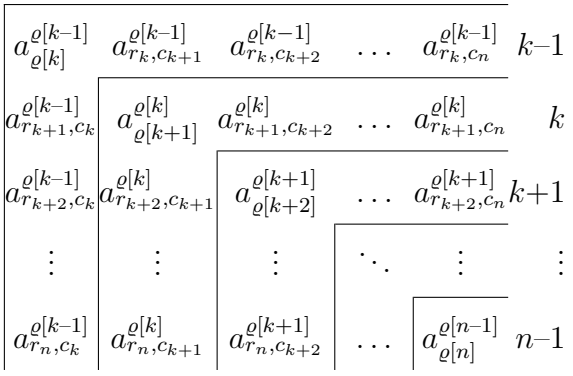
$$f^{\varrho'}_{i,j} = \det\left((A)^{r_1 \ldots r_{k-1}, r_{k+1}, r_k, r_{k+2} \ldots r_{\underline{k}}, r_i}_{c_1 \ldots c_{k-1}, c_{k+1}, c_k, c_{k+2} \ldots c_{\underline{k}}, c_j}\right) = \det\left((A)^{r_1 \ldots r_{\underline{k}}, r_i}_{c_1 \ldots c_{\underline{k}}, c_j}\right) = f^{\varrho}_{i,j};$$

where the second equation applies the fact that swapping two columns and two rows of a matrix does not change its determinant, and the third equation applies the subdeterminant identity of $f^{\varrho}_{i,j}$. □

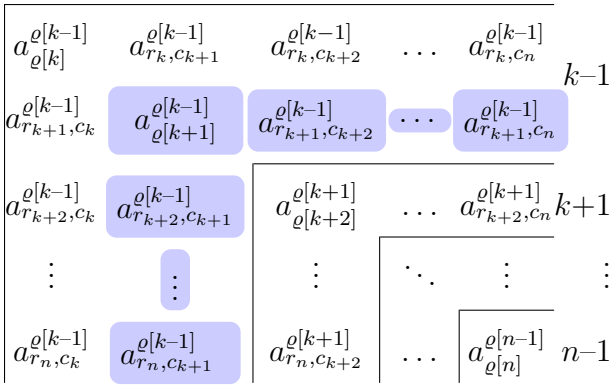
The Lemma 4 requirement that $f^{\varrho}_{k-1,k-1} f^{\varrho}_{k+1,k+1} \neq -f^{\varrho}_{k,k+1} f^{\varrho}_{k+1,k}$ holds automatically whenever $f^{\varrho}_{k,k+1}$ or $f^{\varrho}_{k+1,k}$ are zero since $f^{\varrho}_{k-1,k-1}$ and $f^{\varrho}_{k+1,k+1}$ are nonzero (i.e., the latter two frame entries are pivots). Hence, this requirement is always fulfilled whenever ACPCU cannot be applied. Figure 6.5 provides a graphic description of APDPU; similar to Figure 6.4, it omits frames 0 to k–2 and surrounds the modified entries with a blue box.

Figure 6.5: The Adjacent Pivot Diagonal Permutation Update algorithm

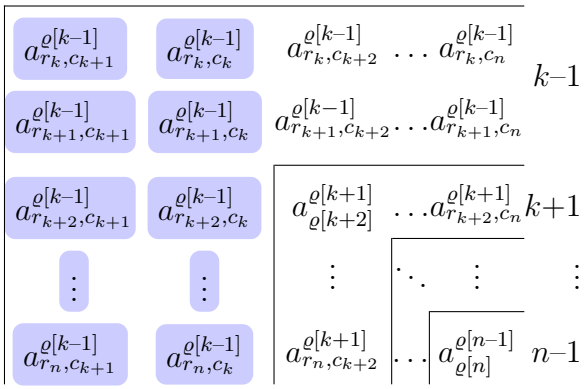
(a) **Input:** F^{ϱ}, k. **Note:** the update $F^{\varrho'}$ is feasible if and only if $f^{\varrho}_{k-1,k-1} f^{\varrho}_{k+1,k+1} \neq -f^{\varrho}_{k,k+1} f^{\varrho}_{k+1,k}$.



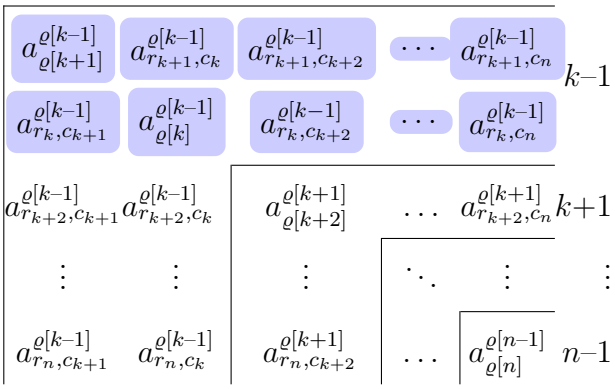
(b) **Step 1:** Backtrack frame k. Notice that frame k–1 expands, frame k vanishes, and the backtracked entries occupy the *inner portion* of frame k–1.



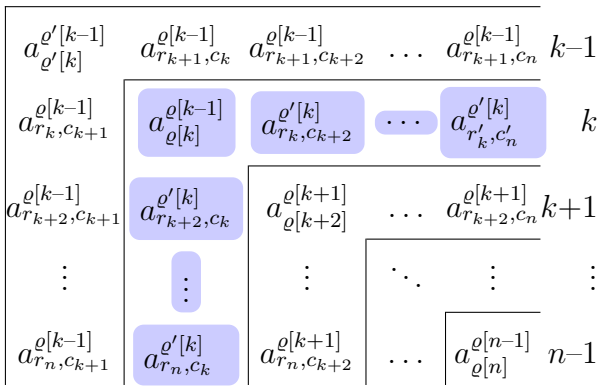
(c) **Step 2:** Permute columns k and k+1 (effect on frames 0 to k–2 not shown).



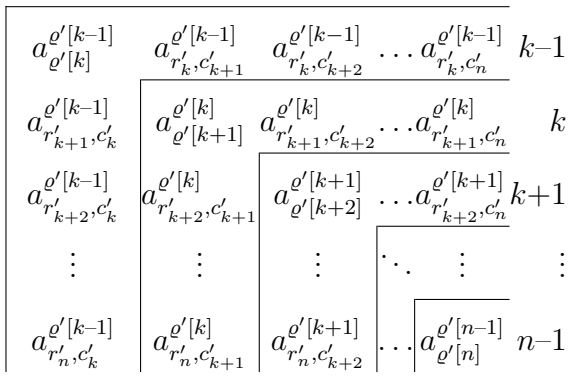
(d) **Step 2:** Permute rows k and k+1 (effect on frames 0 to k–2 not shown).



(e) **Step 3:** Perform an IPGE iteration on the inner portion of expanded frame k–1 using $a^{\varrho[k-1]}_{\varrho[k+1]} = a^{\varrho'[k-1]}_{\varrho'[k]}$ as the new pivot. This reestablishes frame k.



(f) **Output:** $F^{\varrho'}$.



Theorem 6.4.6. *Algorithm APDPU is correct and REF.*

Proof. Equation (6.21) provides the REF formula for backtracking the entries along the vertical segment of frame k; the REF formula for the entries along its horizonal segment is obtained by swapping the subscript row and column indices in the same equation. After permuting columns k and k+1 and rows k and k+1, the entries along frames 0 to k–2 and the outer portion of expanded frame k–1 of the modified matrix match frames 0 to k–1 of $F^{\varrho'}$, respectively. The IPGE formula (i.e., Equation (6.7)) can be applied to obtain the kth frame entries of $F^{\varrho'}$ without roundoff error using the modified entries since,

$$a^{\varrho'[k]}_{r'_i,c'_{k+1}} = \frac{a^{\varrho'[k-1]}_{\varrho'[k]} a^{\varrho'[k-1]}_{r'_i,c'_{k+1}} - a^{\varrho'[k-1]}_{r'_k,c'_{k+1}} a^{\varrho'[k-1]}_{r'_i,c'_k}}{a^{\varrho'[k-2]}_{\varrho'[k-1]}} = \frac{a^{\varrho[k-1]}_{r_{k+1},c_{k+1}} a^{\varrho[k-1]}_{r_i,c_k} - a^{\varrho[k-1]}_{r_{k+1},c_k} a^{\varrho[k-1]}_{r_i,c_{k+1}}}{a^{\varrho[k-2]}_{\varrho[k-1]}}, \text{ and}$$

$$a^{\varrho'[k]}_{r'_{k+1},c'_i} = \frac{a^{\varrho'[k-1]}_{\varrho'[k]} a^{\varrho'[k-1]}_{r'_{k+1},c'_i} - a^{\varrho'[k-1]}_{r'_k,c'_i} a^{\varrho'[k-1]}_{r'_{k+1},c'_k}}{a^{\varrho'[k-2]}_{\varrho'[k-1]}} = \frac{a^{\varrho[k-1]}_{r_{k+1},c_{k+1}} a^{\varrho[k-1]}_{r_k,c_i} - a^{\varrho[k-1]}_{r_{k+1},c_i} a^{\varrho[k-1]}_{r_k,c_{k+1}}}{a^{\varrho[k-2]}_{\varrho[k-1]}};$$

where $i \geq k$+2. Thus, the above formula updates the inner-frame entries of modified frame k–1 to match frame k of $F^{\varrho'}$ using the outer-frame entries of modified frame k–1 (and the corner element of frame k–2). Since frames k+1 to n–1 of F^{ϱ} and $F^{\varrho'}$ are identical by Theorem 6.4.5, no further changes are necessary to yield $F^{\varrho'}$. Hence, APDPU is correct and REF since its constituent operations are REF. □

Corollary 6. *APDPU requires $O(n)$ operations.*

Proof. A backtracking/IPGE operation consists of two multiplications, one division, and one addition. Hence, backtracking frame k and performing 2(n–k)–3 IPGE operations requires $O(n)$ operations. Similarly, permuting rows/columns k and k+1 requires $O(n)$ operations. □

6.5 REF Factorization Updates

The featured REF factorization updates are column addition, deletion, and replacement. They are presented in this order since column replacement is essentially composed of a call to the column addition update followed by a call to the column deletion update. To aid comprehension, the ensuing presentation may be cross-referenced with the corresponding illustrations and discussion of the numerical example showcased in Section 6.6.

Given an existing frame matrix, the REF factorization update algorithms obtain the frame matrix of a respectively modified SLE in $O(n^2)$ operations without accruing roundoff errors. This represents a savings of factor-$O(n)$ operations compared with the $O(n^3)$ operations needed to construct a frame matrix from scratch. As Section 6.3 explained, these algorithms—except for column addition—differ significantly from known factorization update algorithms. In particular they follow the push-and-swap update approach rather than the traditional insert-delete-reduce update approach. The push-and-swap approach explicitly maintains and modifies the links (i.e., recursive relationships) between adjacent frames of F^{ϱ} and avoids additional bit-length growth.

6.5.1 Column Addition

Let $\boldsymbol{a}_{n+1} \in \mathbb{Z}^n$ denote the vector being added to the end of nonsingular $A \in \mathbb{Z}^{n \times n}$ to form the extended integral matrix $\bar{A} := [A \ \boldsymbol{a}_{n+1}]$. Note that $\bar{A}$ has full row-rank based on its simple construction and that the feasibility of ϱ on A extends to $\bar{A}$ because ϱ induces the same basis (and pivot choices) when applied to either matrix. The following theorem introduces and proves the Column Addition Update Algorithm (CAU), which obtains $\bar{F}^{\varrho}$ given F^{ϱ}.

Theorem 6.5.1. *The frame matrix $\bar{F}^{\varrho}$ can be obtained from F^{ϱ} by applying REF*

forward substitution (i.e., Equation (6.2)*) on* $\boldsymbol{a}_{n+1}$ *and adding the resulting vector to the end of* F^{ϱ}*.*

Proof. Since the upper-trapezoidal part of $\bar{F}^{\varrho}$ corresponds to the U matrix of REF-LU of $\bar{A}$ organized according to ϱ, the individual elements of column $n{+}1$ of $\bar{F}^{\varrho}$ equate to IPGE entries as follows:

$$\bar{f}^{\varrho}_{i,n+1} = a^{\varrho[i-1]}_{r_i,n+1} \tag{6.22}$$

Notice the right-hand side is exactly the IPGE entry obtained by applying REF forward substitution with F^{ϱ} as the frame matrix and $\boldsymbol{a}_{n+1}$ as the input right-hand side vector (see Section 6.2). Therefore, letting $\boldsymbol{y}$ be the vector output by REF forward substitution on $\boldsymbol{a}_{n+1}$, this implies $\bar{F}^{\varrho}$ can be expressed as $\bar{F}^{\varrho} = [F^{\varrho} \ \boldsymbol{y}]$. □

Corollary 7. *CAU requires* $O(n^2)$ *operations.*

6.5.2 Column Deletion

Since the column replacement update begins by adding a column to a square frame matrix and ends by deleting one of its columns, say column k, it will be expedient to describe the column deletion update given $A \in \mathbb{Z}^{n\times n+1}$. Thus, letting P be a $n \times n$ row-permutation determined at runtime (see Section 6.3.1), the equation,

$$\bar{A} := P(A \backslash \{A_{(:,k)}\}) \tag{6.23}$$

defines the square integral matrix obtained by deleting $A_{(:,k)}$ from A and then permuting its rows according to P. Recalling that c_i denotes the ith column coordinate in ϱ, when $k \notin \bigcup_{i=1}^{n-1} c_i$, the removal of $A_{(:,k)}$ does not affect the originating pivots of the remaining columns of F^{ϱ} since no elements from column k are actually used as pivots

in the reduced version of IPGE (i.e., the version used to construct F^{ϱ}, as explained in Section 6.2). Hence, in this case, $\bar{F}^{\varrho}$ is simply given by $F^{\varrho}\backslash\{F^{\varrho}_{(:,n)}\}$ or $F^{\varrho}\backslash\{F^{\varrho}_{(:,n+1)}\}$ and $P = I_n$ in Equation (6.23). On the other hand, when $k \in \bigcup_{i=1}^{n-1} c_i$, the removal of $A_{(:,k)}$ modifies the originating pivots and, by extension, the individual values of the kth-iteration matrix of IPGE as well as those of the $(k{+}1)$th to nth-iteration matrices. Nevertheless, it is not necessary to calculate each of these affected iterative matrices—which, in fact, would first require backtracking the entries of frames k to n–1 of $\bar{F}^{\varrho}$ multiple times. Instead, the Column Deletion Update algorithm (CDU) iteratively and efficiently updates the frame links between the affected adjacent pairs of frames in F^{ϱ} to obtain $\bar{F}^{\varrho}$.

CDU is divided into three parts: (1) iteratively pushing column $A_{(:,k)}$ to column position n in F^{ϱ} (i.e., the rightmost basic column); (2) swapping columns n and $n{+}1$ (i.e., the nonbasic column) of the modified frame matrix; and (3) verifying this frame matrix, minus the exiting column, corresponds to a nonsingular coefficient matrix. Steps (c) and (d) in Figure 6.6 offer a visual representation of the first two parts of CDU performed within the column replacement algorithm. It is also possible for CDU to determine if $\bar{A}$ is nonsingular a priori via an additional procedure involving REF backward substitution. However, its description is relocated to the development of the column replacement update, since said procedure is especially suited to that context.

The first part of CDU consists of n–k push steps performed via calls to APCPU or APDPU, which are the elementary factorization updates developed in Section 6.4.2. Given a frame matrix, the requirements for performing at least one of the algorithms is always met since, when the APCPU requirement fails (e.g., $f^{\varrho}_{k,k+1} = 0$ in the first push step), the APDPU requirement is automatically satisfied, as Theorem 6.5.2 explains. APCPU is preferred over APDPU, however, because it requires roughly

half the number of operations as APDPU; if the latter algorithm must be applied, the corresponding row permutation is recorded in the cumulative row-permutation matrix P. After n–k consecutive push steps, the second part of CDU swaps the nth and $(n{+}1)$th columns of the updated frame matrix; this takes the exiting column out of the basis and brings the updated entering column into the basis. The third part of CDU entails simply checking that the updated (n, n)-entry of the working frame matrix is nonzero in order to ensure $\bar{A}$ is nonsingular, which guarantees that it forms a basis and that it admits a proper REF LU factorization.

Theorem 6.5.2. *Algorithm CDU is correct and REF.*

Proof. If the deletion index is nonbasic (i.e., $k = n{+}1$), the result is trivial. Hence, let $k \in \overset{n}{\underset{i=1}{\cup}} c_i$. More specifically, define κ as the IPGE iteration index in which $A_{(:,k)}$ was used as a pivot column to construct F^{ϱ}, that is, $c_\kappa = k$. Additionally, define P^i and Q^i as the $n \times n$ row-permutation and $(n{+}1 \times n{+}1)$ column-permutation matrices, respectively, performed in CDU's ith push step, where $1 \leq i \leq n{-}\kappa$. Each push step involves a call to APCPU or APCPU, meaning its completion provides a working frame matrix, say F^i, corresponding to some row and column permutation of A using pivot sequence ϱ. In particular, at push step i, both APCPU and APDPU swap columns $\kappa{+}i{-}1$ and $\kappa{+}i$ of F^i (i.e., $A_{(:,c_{k+i-1})}$ and $A_{(:,c_{k+i})}$) and, consequently, the cumulative product $Q^1 \dots Q^{n-\kappa} := Q$, is given by:

$$Q = [\boldsymbol{e}_1 \dots \boldsymbol{e}_{\kappa-1} \;\; \boldsymbol{e}_{\kappa+1} \;\; \dots \boldsymbol{e}_n \;\; \boldsymbol{e}_\kappa \;\; \boldsymbol{e}_{n+1}]$$

where $\boldsymbol{e}_i$ is the $(n{+}1)$-length ith elementary vector. In words, Q shifts pivot columns $\kappa{+}1$ to n one position to the left and moves the pivot column originally in frame column position κ to frame column position n. Now, if APCPU is called at push

step i, then $P^i = I_n$ because no rows are permuted; but, if APDPU is called, then rows κ+i–1 and κ+i of F^i are swapped, meaning P^i is given by:

$$P^i = [\boldsymbol{e}_1 \dots \boldsymbol{e}_{\kappa+i-2} \;\; \boldsymbol{e}_{\kappa+i} \;\; \boldsymbol{e}_{\kappa+i-1} \;\; \boldsymbol{e}_{\kappa+i+1} \;\; \dots \boldsymbol{e}_n]$$

where the elementary vectors have length n in this case. Unlike Q, the cumulative product $P^{n-\kappa} \dots P^1 := P$, is not predictable since the specific combination of APCPU and APDPU calls is determined at runtime. In particular, when $f^{i-1}_{\kappa+i-1,\kappa+i}$ is nonzero, Lemma 3 indicates APCPU can be applied, where $F^0 := F^{\varrho}$; while when $f^{i-1}_{\kappa+i-1,\kappa+i}$ is zero, Lemma 4 indicates APDPU can be applied since then $f^{i-1}_{\kappa+i-2,\kappa+i-2} f^{i-1}_{\kappa+i,\kappa+i} \neq -f^{i-1}_{\kappa+i-1,\kappa+i} f^{i-1}_{\kappa+i,\kappa+i-1} = 0$ (the left-hand side in the first equation is the product of two nonzero pivots). Hence, at least one of these two push step subroutines can always be applied. This implies that, at the conclusion of all the push steps, the frame matrix of $P^{n-\kappa} \dots P^1 A Q^1 \dots Q^{n-\kappa} = PAQ$ associated with ϱ, i.e., $F^{n-\kappa}$, is obtained without accruing roundoff errors. From Equation (6.22), notice that the originating pivots of the nth and (n+1)th columns of F^{ϱ} are identical along each corresponding row—i.e., they lie in the same frame—since they are given by $f^{\varrho[i-1]}_{i,n}$ and $f^{\varrho[i-1]}_{i,n+1}$, respectively, for $1 \leq i \leq n$. Similarly, because APCPU and APDPU update the originating pivots of full frames, the entries of columns n and n+1 of $F^{n-\kappa}$ along matching rows indices are connected to the same originating pivots. This means $F^{n-\kappa}_{(:,n)}$ can be swapped with $F^{n-\kappa}_{(:,n+1)}$ and then deleted to yield the $n \times n$ frame matrix $\bar{F}^{\varrho}$, which is properly defined as long as its underlying coefficient matrix $\bar{A}$ is nonsingular. The latter condition is verified by ensuring that the post-swap (n, n)-entry of the working frame matrix is nonzero due to the fact that $\bar{f}^{\varrho}_{n,n} = \det(\bar{A})$ by Equation (6.10). Since $\bar{A}$ is obtained by removing column k of A, shifting columns k+1 to n+1 one position left, and then permuting rows according to P, CDU obtains $\bar{F}^{\varrho}$ or determines $\bar{A}$ is singular (i.e., it

does not allow a proper REF LU factorization) without accruing roundoff errors. □

Corollary 8. *CDU requires $O(n^2)$ operations.*

Proof. APCPU and APDPU require $O(n)$ operations by Corollaries 5 and 6. Hence, performing n–k push steps requires $O(n^2)$ operations. Additionally, swapping columns n and n+1 and verifying the underlying post-swap square coefficient matrix is non-singular require $O(n)$ and $O(1)$ operations, respectively. □

6.5.3 Column Replacement

Let $\boldsymbol{a}_{n+1} \in \mathbb{Z}^n$ be a replacement nonzero-vector for an exiting column $A_{(:,c_k)}$ in $A \in \mathbb{Z}^{n\times n}$ determined at runtime, where $1 \leq k \leq n$, and let P be an $n \times n$ row-permutation matrix. Then, the equation,

$$\bar{A} := P[(A - A_{(:,c_k)}) \ \boldsymbol{a}_{n+1}] \tag{6.24}$$

defines the matrix obtained after shifting columns c_k+1 to n of A one position left (i.e., erasing the column originally in position c_k), inserting $\boldsymbol{a}_{n+1}$ in the vacated nth column position, and permuting its rows according to P (also determined at runtime). The Column Replacement Update algorithm (CRU) provides a way to obtain $\bar{F}^{\varrho}$ given F^{ϱ} that differs from the traditional delete-then-insert update methodology discussed in Section 6.3.1. At each step, the algorithm shifts $A_{(:,k)}$ one place to the right in the working frame matrix and modifies the affected frames appropriately; the last shift is performed by swapping columns n and n+1 of this matrix. For this reason, its mechanism is aptly characterized as pushing the exiting column until it can be swapped out of the basis.

CRU consists of three subroutines executed in succession: (1) call *CAU* given $\boldsymbol{a}_{n+1}$, (2) determine a candidate set of exiting columns and select a column index,

say c_k, from this set, and (3) call CDU given column index c_k.

Determining the candidate set of exiting columns and then selecting an individual column from this set ensures that $\bar{A}$ is nonsingular, thereby avoiding the need for CDU to verify said property. Let $A_{(:,J)}$ be the exiting-column candidate set, which is induced by index set $J \subseteq \{1 \dots n\}$. To determine the elements of J, denote $\boldsymbol{y}$ as the column vector added by CAU (i.e., $\boldsymbol{y}$ is the result of REF forward substitution using F^{ϱ} on vector $\boldsymbol{a}_{n+1}$). Additionally, denote $\boldsymbol{x'}$ as the result of REF backward substitution using F^{ϱ} on scaled vector $\boldsymbol{y'} = \det(A)\boldsymbol{y} = f^{\varrho}_{n,n}\boldsymbol{y}$ (Section 6.2 explains why $\boldsymbol{y}$ must be scaled by $\det(A)$). The set J is composed of the indices of the nonzero entries of $\boldsymbol{x'}$; formally, $J := \{j \mid x'_j \neq 0\}$. To keep the focus on the factorization update process, we assume the index $c_k \in J$ of the exiting variable is chosen arbitrarily. In practice, the simplex algorithm determines this index using the ratio test.

Figure 6.6 offers a step-by-step description and accompanying visual representation of CRU. The subfigures within Figure 6.6 omit frames 0 to k–2 due to the general inactivity of their entries in the algorithm. Moreover, this visual representation assumes that APCPU is performed at each push step and that ϱ is fixed according to the sequence given by Assumption 2, that is, $\varrho[k] := (k, k)$ for $1 \leq k \leq n$. We make these assumptions for two reasons. First, they complement the graphic outline of CRU given back in Section 6.3.2 and, as a result, Figure 6.6 fills in the values of the then-unknown matrix entries of Figure 6.2. Second, by fixing ϱ's elements, the evolution of each entry's originating pivot is displayed in terms of its changing actual coordinates rather than in terms of ϱ, ϱ', and similarly abstracted pivot sequences.

Theorem 6.5.3. *Algorithm CRU is correct and REF.*

Proof. Since CRU is composed of a call to CAU followed by a call to CDU, the

Figure 6.6: The Column Replacement Update algorithm

(a) **Subroutine 1 (CAU):** Perform REF forward substitution given $\boldsymbol{a}_{n+1}$ and append the resulting vector to the right of F^{ϱ}; denote the output $(n+1)$th column (i.e., the nonbasic column) as $\boldsymbol{y}$. **Subroutine 2 (Exiting column selection):** Perform REF backward substitution on $\boldsymbol{y}' = f^{\varrho}_{n,n}\boldsymbol{y}$ to obtain $\boldsymbol{x}'$ and select the exiting column index, say k, from the nonzero elements of $\boldsymbol{x}'$ (the exiting column k is shaded).

k	$k{+}1$	$k{+}2$	$\dots$	n	$n{+}1$	
$a^{(k-1,k-1)}_{k,k}$	$a^{(k-1,k-1)}_{k,k+1}$	$a^{(k-1,k-1)}_{k,k+2}$	$\dots$	$a^{(k-1,k-1)}_{k,n}$	$y_k = a^{(k-1,k-1)}_{k,n+1}$	$k{-}1$
$a^{(k-1,k-1)}_{k+1,k}$	$a^{(k,k)}_{k+1,k+1}$	$a^{(k,k)}_{k+1,k+2}$	$\dots$	$a^{(k,k)}_{k+1,n}$	$y_{k+1} = a^{(k,k)}_{k+1,n+1}$	k
$a^{(k-1,k-1)}_{k+2,k}$	$a^{(k,k)}_{k+2,k+1}$	$a^{(k+1,k+1)}_{k+2,k+2}$	$\dots$	$a^{(k+1,k+1)}_{k+2,n}$	$y_{k+2} = a^{(k+1,k+1)}_{k+2,n+1}$	$k{+}1$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$a^{(k-1,k-1)}_{n,k}$	$a^{(k,k)}_{n,k+1}$	$a^{(k+1,k+1)}_{n,k+2}$	$\dots$	$a^{(n-1,n-1)}_{n,n}$	$y_n = a^{(n-1,n-1)}_{n,n+1}$	$n{-}1$

(b) **Subroutine 3 (CDU column pushes):** Perform n–k push steps via the APCPU or APDPU algorithms (the first push step—performed via APCPU—is shown): APCPU is called if the pivot-row-adjacent replacement entry (e.g., $a^{(k-1,k-1)}_{k,k+1}$) is nonzero; APDPU is called otherwise since its requirements are automatically satisfied when APCPU cannot be applied.

$k{+}1$	k	$k{+}2$	$\dots$	n	$n{+}1$	
$a^{(k-1,k-1)}_{k,k+1}$	$a^{(k-1,k-1)}_{k,k}$	$a^{(k-1,k-1)}_{k,k+2}$	$\dots$	$a^{(k-1,k-1)}_{k,n}$	$a^{(k-1,k-1)}_{k,n+1}$	$k{-}1$
$a^{(k-1,k-1)}_{k+1,k+1}$	$a^{(k,k+1)}_{k+1,k}$	$a^{(k,k+1)}_{k+1,k+2}$	$\dots$	$a^{(k,k+1)}_{k+1,n}$	$a^{(k,k+1)}_{k+1,n+1}$	k
$a^{(k-1,k-1)}_{k+2,k+1}$	$a^{(k,k+1)}_{k+2,k}$	$a^{(k+1,k)}_{k+2,k+2}$	$\dots$	$a^{(k+1,k)}_{k+2,n}$	$a^{(k+1,k)}_{k+2,n+1}$	$k{+}1$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$a^{(k-1,k-1)}_{n,k+1}$	$a^{(k,k+1)}_{n,k}$	$a^{(k+1,k)}_{n,k+2}$	$\dots$	$a^{(n-1,n-1)}_{n,n}$	$a^{(n-1,n-1)}_{n,n+1}$	$n{-}1$

(c) **Subroutine 3, continued (CDU column swap):** Swap the exiting column—which is in the rightmost basic column position after performing the column pushes of Step (c)—with the modified nonbasic column. The resulting matrix (shown), minus the shaded column, corresponds to $\bar{F}^{\varrho}$, the frame matrix of $\bar{A}$

$k{+}1$	$k{+}2$	$k{+}3$	$\dots$	$n{+}1$	k	
$a^{(k-1,k-1)}_{k,k+1}$	$a^{(k-1,k-1)}_{k,k+2}$	$a^{(k-1,k-1)}_{k,k+3}$	$\dots$	$a^{(k-1,k-1)}_{k,n+1}$	$a^{(k-1,k-1)}_{k,k}$	$k{-}1$
$a^{(k-1,k-1)}_{k+1,k+1}$	$a^{(k,k+1)}_{k+1,k+2}$	$a^{(k,k+1)}_{k+1,k+3}$	$\dots$	$a^{(k,k+1)}_{k+1,n+1}$	$a^{(k,k+1)}_{k+1,k}$	k
$a^{(k-1,k-1)}_{k+2,k+1}$	$a^{(k,k+1)}_{k+2,k+2}$	$a^{(k+1,k+2)}_{k+2,k+3}$	$\dots$	$a^{(k+1,k+2)}_{k+2,n+1}$	$a^{(k+1,k+2)}_{k+2,k}$	$k{+}1$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$a^{(k-1,k-1)}_{n,k+1}$	$a^{(k,k+1)}_{n,k+2}$	$a^{(k+1,k+2)}_{n,k+3}$	$\dots$	$a^{(n-1,n)}_{n,n+1}$	$a^{(n-1,n)}_{n,k}$	$n{-}1$

algorithm is REF. For the same reason, at its conclusion the algorithm yields a proper frame matrix that corresponds to the matrix $\bar{A}$ defined by Equation (6.24), provided that the rule for selecting the exiting variable guarantees that the underlying updated basis matrix is nonsingular. Hence, it suffices to prove the validity of the said rule to complete the proof.

By definition of IPGE and the REF factorization framework, the result of successively performing REF forward and backward substitution on $\boldsymbol{a}_{n+1}$ is the REF vector $\boldsymbol{x}' = \det(A)\boldsymbol{x}$, where $\boldsymbol{x}$ is the solution of the SLE $A\boldsymbol{x} = \boldsymbol{a}_{n+1}$. Since A's columns form a basis, this means $\boldsymbol{a}_{n+1}$ can be stated as the following linear combination:

$$\boldsymbol{a}_{n+1} = \sum_{j=1}^{n} x_j \boldsymbol{a}_j$$

where $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ are the individual columns of A. Having established this relationship, the standard argument for replacing a vector in a basis by another vector (e.g., see [11]) states that $x_k \neq 0$ is a necessary and sufficient condition for the set of vectors $\boldsymbol{a}_1, \ldots \boldsymbol{a}_{k-1}, \boldsymbol{a}_{k+1}, \ldots \boldsymbol{a}_{n+1}$ to be linearly independent. Moreover, $x'_k = 0$ if and only if $x = 0$ since A is nonsingular (i.e., $\det(A) \neq 0$). □

Corollary 9. *CRU requires $O(n^2)$ operations.*

Proof. CRU's first and third subroutines, CAU and CDU, respectively, require $O(n^2)$ operations by Corollaries 7 and 8. Moreover, its second subroutine involves performing REF backward substitution on an n-length vector, which requires $O(n^2)$ operations (see Section 4.3), and then selecting an index from the nonzero elements of the resulting vector, which requires $O(n)$ operations. □

Theorem 6.5.4. *The REF column addition, column deletion, and column replacement updates achieve the computational savings expected of traditional LU factoriza-*

tion updates and they do not lead to further growth in the bit-length required by their matrix entries.

Proof. The expected savings stem from Corollaries 7, 8, and 9. To prove the second statement, notice that all the iterative frame matrices obtained in these three algorithms correspond to a REF-LU factorization of a row or column permutation of A or $\bar{A}$. Moreover, the elementary updates and shortcuts performed within column deletion and replacement involve backtracking IPGE entries, changing their originating pivots, or flipping their signs. Hence, the bit-length of every iterative matrix entry involved in all the featured column update algorithm is bounded by the maximum bit-length required by the REF factorization process (see IPGE Property 3), meaning no further bit-length growth is required. □

Corollary 10. *The worst-case case computational complexity (WCC) of the featured REF-LU factorization updates is given by:*

$$\begin{aligned} \text{WCC(REF-LU updates)} &= O(n^2(\omega_{\max} \log \omega_{\max} \log\log \omega_{\max})) && (6.25) \\ &= O\left(n^3 \max(\log^2 n \log\log n, \log^2 \sigma \log\log \sigma)\right). && (6.26) \end{aligned}$$

Proof. The expression in the innermost parentheses of Equation (6.25) represents the cost of multiplying/dividing two integers of bit-length $\omega_{\max}$ (the IPGE maximum world-length), which bounds the bit-length of the individual matrix entries in all the REF updates by Theorem 6.5.4; these costs are derived from the use of Fast Fourier transform techniques. The quantity outside the innermost parentheses represents the required number of operations of the update algorithms by Corollaries 7, 8, and 9. □

6.6 Illustrative Example of REF Column Replacement

This subsection depicts the step-by-step application of the REF column replacement update algorithm (CRU) via a numerical example in order to reinforce many of the concepts heretofore discussed. For this purpose, fix A and $\bar{A}$ to be the following nonsingular integral (4×4)-matrices:

$$A = \begin{bmatrix} 3 & 11 & 8 & 7 \\ 5 & 2 & 3 & 5 \\ 6 & -7 & -2 & 1 \\ 7 & 10 & -2 & -6 \end{bmatrix} \qquad \tilde{A} = \begin{bmatrix} 3 & 8 & 7 & 1 \\ 5 & 3 & 5 & 4 \\ 6 & -2 & 1 & 7 \\ 7 & -2 & -6 & 11 \end{bmatrix}.$$

Additionally, define the vector $\boldsymbol{a}_5 := \bar{A}_{(:,4)}$ so that the relationship between A and $\bar{A}$ can be conveniently characterized by the equation:

$$\bar{A} := P[(A - A_{(:,2)}) \ \boldsymbol{a}_5], \tag{6.27}$$

which fits the form given by Equation (6.24) with $P = I_4$. Since A and $\bar{A}$ are nonsingular, their frame matrices—i.e., the merged lower-triangular and upper-triangular matrices of their REF LU factorizations—are well defined for some pivot sequence ϱ; they are denoted accordingly as F^{ϱ} and $\bar{F}^{\varrho}$ and their contents are as follows:

$$F^{\varrho} = \begin{array}{c|ccc|l} 3 & 11 & 8 & 7 & \kappa = 0 \\ \hline 5 & -49 & -31 & -20 & \kappa = 1 \\ \hline 6 & -87 & -17 & 57 & \kappa = 2 \\ \hline 7 & -47 & 527 & 884 & \kappa = 3 \end{array} \qquad \bar{F}^{\varrho} = \begin{array}{c|ccc|l} 3 & 8 & 7 & 1 & \kappa = 0 \\ \hline 5 & -31 & -20 & 7 & \kappa = 1 \\ \hline 6 & -54 & 43 & -29 & \kappa = 2 \\ \hline 7 & -62 & 279 & -89 & \kappa = 3 \end{array}$$

when $\varrho = \{(1,1),(2,2),(3,3),(4,4)\}$ is the pivot (coordinate) sequence applied by the IPGE runs on both A and $\bar{A}$. Constructing $\bar{F}^{\varrho}$ (or F^{ϱ}) from scratch requires $O(n^3)$ operations. However, assuming F^{ϱ} has been calculated, CRU obtains $\bar{F}^{\varrho}$ in $O(n^2)$ operations while keeping the same bit-length upper bound as the original REF factorization, as the ensuing paragraphs illustrate.

6.6.1 Column Addition Subroutine

The first step is to add the entering column to F^{ϱ} and to select a valid exiting column index k; to enhance clarity, we skip showing the latter process—which simply involves performing REF forward and backward substitution on the entering column—and assume that $k = 2$. The entering column is added by performing REF forward substitution (i.e., Equation (6.2)) using F^{ϱ} as the frame matrix and $\boldsymbol{a}_5$ as the right-hand side vector, and then appending the updated column. The output is actually the frame matrix of $[A \ \boldsymbol{a}_5]$ associated with ϱ, denoted as $\hat{F}^{\varrho}$. Hence, we obtain the following expanded frame matrix:

$$\hat{F}^{\varrho} = \begin{array}{c|cccc|l} 3 & 11 & 8 & 7 & 1 & \kappa = 0 \\ \hline 5 & -49 & -31 & -20 & 7 & \kappa = 1 \\ \hline 6 & -87 & -17 & 57 & -42 & \kappa = 2 \\ \hline 7 & -47 & 527 & 884 & -561 & \kappa = 3 \end{array},$$

where the exiting column is shaded and the updated entering column is right-adjacent to the dotted line. CRU now moves on to the column deletion phase (i.e., the push steps).

6.6.2 Column Deletion Subroutine

The push-and-swap update approach gradually drives the exiting column out of the basis via individual adjacent pivot-column (APCPU) or adjacent pivot-diagonal (APDPU) permutation updates. To match the desired frame matrix of this example, the pushes must be performed via two calls to APCPU; Figure 6.7 zooms in on the first APCPU update to highlight the use of the auxiliary REF operations introduced in Section 6.4.1. At the conclusion of this first call to ACPU, the working frame matrix becomes:

$$\hat{F}^{\varrho'} = \begin{array}{c|cc|cc|l} 3 & 8 & 11 & 7 & 1 & \kappa = 0 \\ \hline 5 & -31 & -49 & -20 & 7 & \kappa = 1 \\ \hline 6 & -54 & 17 & 43 & -29 & \kappa = 2 \\ \hline 7 & -62 & -527 & -884 & 561 & \kappa = 3 \end{array},$$

where $\varrho' = \{(1,1),(2,\underline{3}),(3,\underline{2}),(4,4)\}$ is the pivot sequence that would be required by an IPGE run on $[A\ \boldsymbol{a}_5]$ to obtain the new frame matrix; the permuted pivot indices are underlined in ϱ'. Notice that only the entries on the strictly vertical part of frame 1 and on the strictly horizontal part of frame 2 change in value; the remaining frame-2 entries and all frame-3 entries change only in sign. Similarly, the

Figure 6.7: The first ACPCPU push step applied to a numerical example

(a) **Input:** $\hat{F^{\varrho}}, k = 2$. **Note:** the update is feasible since the entry $f^{\hat{\varrho}}_{2,3}$ (boxed in blue) is nonzero. (During the update process, the working matrix may not fit the strict definition of a frame matrix. Hence, at times, an entry of the working matrix may be denoted simply as $f_{i,j}$ to draw this distinction).

3	11	8	7	1	$\kappa = 0$
5	−49	−31	−20	7	$\kappa = 1$
6	−87	−17	57	−42	$\kappa = 2$
7	−47	527	884	−561	$\kappa = 3$

(b) **Step 1:** Backtrack entries $f^{\hat{\varrho}}_{3,3} = a^{2,2}_{3,3}$ and $f^{\hat{\varrho}}_{4,3} = a^{2,2}_{4,3}$ to obtain $a^{1,1}_{3,3}$ and $a^{1,1}_{4,3}$, as follows: $f_{3,2} \leftarrow a^{1,1}_{3,3} = \frac{3(-17)+(-31)(-87)}{-49} = \frac{2646}{-49} = -54$; $f_{4,2} \leftarrow a^{1,1}_{4,3} = \frac{3(527)+(-31)(-47)}{-49} = \frac{3038}{-49} = -62$. (The backtracked entries are placed in column 2, as the boxing in blue indicates).

3	11	8	7	1	$\kappa = 0$
5	−49	−31	−20	7	$\kappa = 1$
6	−54	−17	57	−42	$\kappa = 2$
7	−62	527	884	−561	$\kappa = 3$

(c) **Step 2:** Perform RwSOP on $f^{\hat{\varrho}}_{3,4} = a^{2,2}_{3,4}$ and $f^{\hat{\varrho}}_{3,5} = a^{2,2}_{3,5}$ to obtain $a^{2,3}_{3,4}$ and $a^{2,3}_{3,5}$, as follows: $f_{3,4} \leftarrow a^{2,3}_{3,4} = \frac{-31(57)-(-20)(-17)}{-49} = \frac{-2107}{-49} = 43$; $f_{3,5} \leftarrow a^{2,3}_{3,5} = \frac{-31(-42)-(-7)(-17)}{-49} = \frac{1421}{-49} = -29$. (The updated entries are boxed in blue).

3	11	8	7	1	$\kappa = 0$
5	−49	−31	−20	7	$\kappa = 1$
6	−54	−17	43	−29	$\kappa = 2$
7	−62	527	884	−561	$\kappa = 3$

(d) **Step 3:** Permute columns 2 and 3 of frames 0 to 1. (The affected entries are boxed in blue).

3	8	11	7	1	$\kappa = 0$
5	−31	−49	−20	7	$\kappa = 1$
6	−54	−17	43	−29	$\kappa = 2$
7	−62	527	884	−561	$\kappa = 3$

(e) **Step 4:**Change the sign of the entries on the strictly vertical part of frame 2 and the entries in frame 3. (The affected entries are boxed in blue).

3	8	11	7	1	$\kappa = 0$
5	−31	−49	−20	7	$\kappa = 1$
6	−54	17	43	−29	$\kappa = 2$
7	−62	−527	−884	561	$\kappa = 3$

outcome of the second call to APCPU gives:

$$\hat{F}^{\varrho'} = \begin{array}{c|cccc|c} 3 & 8 & 7 & 11 & 1 & \kappa = 0 \\ \hline 5 & -31 & -20 & -49 & 7 & \kappa = 1 \\ \hline 6 & -54 & 43 & 17 & -29 & \kappa = 2 \\ \hline 7 & -62 & 279 & 884 & -89 & \kappa = 3 \end{array},$$

where $\varrho' = \{(1,1),(2,3),(3,\underline{4}),(4,\underline{2})\}$ is the updated pivot sequence. As the above frame matrices illustrate, each APCPU call interchanges two adjacent pivot columns in the frame matrix and then efficiently propagates the effect to all subsequent frames. An alternative interpretation is that, keeping the same pivot sequence ϱ throughout the update process, these operations swiftly transition from the frame matrix of $[A_{(:,1)}\ A_{(:,2)}\ A_{(:,3)}\ A_{(:,4)}]$ to that of $[A_{(:,1)}\ A_{(:,3)}\ A_{(:,2)}\ A_{(:,4)}]$ and then to that of $[A_{(:,1)}\ A_{(:,3)}\ A_{(:,4)}\ A_{(:,2)}]$. Hence, the first two columns of $\hat{F}^{\varrho'}$ and $\bar{F}^{\varrho}$ and the first three columns of $\hat{F}^{\varrho'}$ and $\bar{F}^{\varrho}$ match after the first and second calls to APCPU, respectively.

6.6.3 Column Swap Subroutine

After the exiting column has been pushed to the fourth column position (i.e., the rightmost basic position), its rows lie along the same frames as those of the entering column. Hence, the columns are swapped safely, after which the working

frame matrix becomes:

$$\begin{array}{cccc|c|l} 3 & 8 & 7 & 1 & 11 & \kappa = 0 \\ \hline 5 & -31 & -20 & 7 & -49 & \kappa = 1 \\ \hline 6 & -54 & 43 & -29 & 17 & \kappa = 2 \\ \hline 7 & -62 & 279 & -89 & 884 & \kappa = 3 \end{array} .$$

The resulting matrix, minus the shaded column, equals the target frame matrix $\bar{F}^{\varrho}$.

6.6.4 Commentary on Required Bit-Length Upper Bound

All the entries obtained throughout CRU correspond to individual entries of some IPGE run on A. Hence, the algorithm retains the same bit-length upper bound as the original REF factorization of A (see IPGE Property 3) or, in other words, it does not lead to additional entry growth. Conversely, applying the simple REF variant of the Bartels-Golub update on the upper-triangular part of F^{ϱ} yields, after $O(n^2)$ operations, the following "inflated" upper-triangular matrix:

$$\begin{bmatrix} 3 & 8 & 7 & 1 \\ 0 & -31 & -20 & 7 \\ 0 & 0 & -2107 & 1421 \\ 0 & 0 & 0 & -74137 \end{bmatrix} .$$

Notice this matrix does not match the upper triangular part of $\bar{F}^{\varrho}$ and that its entries along row 3 and 4 are significantly larger. In fact, this and similar naive updates lead to the bit-length upper bound given by Equation (6.6), which represents a significant magnification from the REF LU bit-length upper bound given by Equation (2.14).

As Section 6.3.1 explains, another downside to applying this naive algorithm is that it is not straightforward to update the lower-triangular part of F^{ϱ} to retain the special structure of the REF LU factorizations.

6.7 Extensions to Row Updates and to the REF Cholesky Factorization

Extending the featured framework to perform row factorization updates does not require deriving complementary REF row addition, deletion, and replacement updates. This is because applying the column updates on $(F^{\varrho})^T$, the transposed frame matrix of A, and then transposing the updated frame matrix produces the same result. To prove this, let $A^T := \left(a^T_{i,j}\right)$ be the transpose of A and let ϱ' be the pivot sequence that swaps the row and column coordinates of feasible pivot sequence ϱ, that is, $\varrho'[k] := (r'_k, c'_k) = (c_k, r_k)$, for $1 \leq k \leq n$. The following correspondence can be established between the IPGE run on A associated with ϱ and the IPGE run on A^T associated with ϱ'.

Lemma 5. *For $i, j, k \in \mathbb{Z}$, such that $0 \leq k \leq n$ and $1 \leq i, j \leq n$, with $i \neq r'_k = c_k$, the following equivalence holds:*

$$a^{T\varrho'[k]}_{i,j} = a^{\varrho[k]}_{j,i}.$$

Proof. We prove the correspondence by induction on k. For this purpose, let $h \in \mathbb{Z}$ be such that $0 < h \leq n$.

Base case: $k = 0$. The result is trivial since $a^{T\varrho'[0]}_{i,j} = a^T_{i,j} = a_{j,i} = a^{\varrho[0]}_{j,i}$ by definition of a transpose matrix.

Inductive step: $k = h$. Assume the result holds for $k = 0 \ldots h$–1. Applying

the definition of IPGE, given by Equation (6.7), we have:

$$a_{i,j}^{T\varrho'[h]} = \frac{a_{\varrho'[h]}^{T\varrho'[h-1]} a_{i,j}^{T\varrho'[h-1]} - a_{r'_h,j}^{T\varrho'[h-1]} a_{i,c'_h}^{T\varrho'[h-1]}}{a_{\varrho'[h-1]}^{T\varrho'[h-2]}} \tag{6.28}$$

$$= \frac{a_{c_h,r_h}^{T\varrho'[h-1]} a_{i,j}^{T\varrho'[h-1]} - a_{c_h,j}^{T\varrho'[h-1]} a_{i,r_h}^{T\varrho'[h-1]}}{a_{c_{h-1},r_{h-1}}^{T\varrho'[h-2]}} \tag{6.29}$$

$$= \frac{a_{r_h,c_h}^{\varrho[h-1]} a_{j,i}^{\varrho[h-1]} - a_{j,c_h}^{\varrho[h-1]} a_{r_h,i}^{\varrho[h-1]}}{a_{r_{h-1},c_{h-1}}^{\varrho[h-2]}} \tag{6.30}$$

$$= \frac{a_{\varrho[h]}^{\varrho[h-1]} a_{j,i}^{\varrho[h-1]} - a_{r_h,i}^{\varrho[h-1]} a_{j,c_h}^{\varrho[h-1]}}{a_{\varrho[h-1]}^{\varrho[h-2]}} \tag{6.31}$$

$$= a_{j,i}^{\varrho[h]}, \tag{6.32}$$

where Equation (6.29) substitutes the elements of $\varrho'[h]$ with their corresponding elements in $\varrho[h]$; where Equation (6.30) applies the inductive hypothesis to each (h–1)th and (h–2)th iteration entry; where Equation (6.31) reorganizes factors and implements the succinct form of the generalized IPGE notation; and, where Equation (6.32) results from the definition of $a_{j,i}^{\varrho[h]}$. Thus, the result holds for $k = h$. Since i, j and h, such that $1 \leq i, j \leq n$ and $0 < h \leq n$, were chosen arbitrarily, the result holds true for $k = 0 \ldots n$ and for all i and j. □

Theorem 6.7.1. *Applying column updates on $(F^{\varrho})^T$ is equivalent to applying row updates on F^{ϱ}.*

Proof. The correspondence established by Lemma 5 implies that the IPGE run on A^T associated with ϱ', denoted as $A^{T\varrho'}$, induces a frame matrix $F^{T\varrho'}$ that is exactly the transpose of F^{ϱ}. However, through REF substitution, $F^{T\varrho'}$ solves the SLE $A^T\boldsymbol{x} = \boldsymbol{b}$ or, equivalently, $\boldsymbol{x}^T A = \boldsymbol{b}^T$ (i.e., a linear system of row combinations of A). It follows that updating the columns of $F^{T\varrho'} = (F^{\varrho})^T$, is equivalent to updating the rows of F^{ϱ}. □

In order to describe how the REF factorization update algorithms extend to the REF Cholesky factorization (REF-Ch), we remark that it does not make sense to update the kth column of a Cholesky factorization without updating its kth row concurrently since the factorization must preserve symmetry. For this reason, Cholesky factorization updates are implicitly performed as symmetric row-column pairs when a single row or individual column is updated. For instance, the column addition of $\boldsymbol{a}_{n+1}$ is implicitly accompanied by the row addition of $(\boldsymbol{a}_{n+1})^T$. Column deletion and replacement are analogously paired with their symmetric row counterparts.

Fix A and $\bar{A}$ to be symmetric positive definite so that each matrix admits a REF Cholesky factorization. Since REF-Ch is symmetric, the same holds true for the corresponding frame matrices F^{ϱ} and $\bar{F}^{\varrho}$, whose upper triangular sections do not need to be explicitly stored since they equal the transpose of the lower triangular sections (i.e., F^{ϱ} and $\bar{F}^{\varrho}$ are composed of only L and $\bar{L}$, respectively). Similarly, in order to maintain the storage advantage of REF-Ch, the intermediary frame matrices attained in the process of updating F^{ϱ} into $\bar{F}^{\varrho}$ must preserve symmetry. This means that the adjacent pivot column permutation update cannot be performed since it changes the horizontal portion of a frame via a row-wise switch of originating pivot, but it changes only the signs of the vertical portion of the same frame through backtracking, thereby creating an asymmetric frame structure. This applies to the adjacent pivot row permutation update as well since it does the transpose of this process. Conversely, the adjacent pivot diagonal permutation update is valid since it performs a change of originating pivot and backtracking operations to the full frame—in implementation, however, the backtrack/change of originating pivot operations are performed explicitly on the vertical part of the frame and implicitly on its horizontal part.

Through the implementation of the adjacent pivot diagonal permutation update,

the same updates performed on column k of REF-Ch are simultaneously performed on row k, as expected by the Cholesky update process. Essentially, the REF-Ch deletion and replacement updates push row k and column k out of F^{ϱ} simultaneously in the process of obtaining $\bar{F}^{\varrho}$. With respect to the column-row replacement update, once the kth row and column are pushed to their nth basic positions and swapped with their updated nonbasic counterparts, the REF update process makes it easy to determine if $\bar{A}$ is in fact positive definite. This entails checking that the post-swap (n, n)-entry of the working frame matrix is nonzero since this entry equals $\det(\bar{A})$ at this stage (see the proof to Theorem 6.5.2).

6.8 Conclusions

This section explains that applying the traditional delete-insert-reduce update approach to the REF factorizations induces inefficient algorithms in terms of bit-length growth and increased computational effort. In short, this approach is inadequate because it neglects the special structure of the REF factorizations. Conversely, the presented push-and-swap update approach maintains and updates the recursive relationships between the consecutive rows/columns of the REF factorization. Through this approach, the featured REF LU column addition, deletion, and replacement updates achieve the operations savings expected of factorization update algorithms, which they are able to accomplish by avoiding additional growth in the bit-length required by their matrix entries. The current work also proves that the complementary REF row updates can be performed via the REF column updates, and it discusses special considerations for updating the REF Cholesky factorization.

The motivation for developing and continuing to improve the REF factorization framework is based on its potential to enhance current exact simplex-based linear programming (LP) solvers and mixed-integer programming solvers. Indeed, a major

goal of our ongoing research is to craft algorithms that can be ultimately integrated into existing LP solvers in order to equip them with efficient tools for avoiding some of the inconsistent solver outputs discussed in Section 1. A prospective implementation of the algorithms herein presented would follow a similar blueprint as the development of the exact LP solvers within `QSopt_ex` [5] and `SoPlex` [88, 89]. In particular, as Sections 3.3.3 and 3.3.4 explain, these solvers implement improved versions of the exact rational arithmetic validation algorithm pioneered by Dhiflaoui et al. [20], which works by applying an exact rational arithmetic LU factorization to verify the validity of the floating-point simplex solution to a given rational LP. Whenever the basic solution validation process determines the solver-provided solution is invalid, the exact LP solvers restart the simplex algorithm from the last known basis. They induce more simplex iterations by augmenting their working precision or by adjusting algorithmic settings that lead to better solutions, which must in turn be put through the validation process. Hence, the REF factorization update algorithms would be implemented to avoid having to construct an exact LU factorization at each subsequent validation step.

7. CONCLUSIONS

This dissertation introduces an efficient factorization framework for solving systems of linear equations (SLEs) exactly. In particular, it develops the roundoff-error-free (REF) LU and Cholesky factorizations and the REF forward and backward substitution algorithms, which combine to calculate exact solutions to SLEs efficiently. This tool set provides an efficient alternative to the exact rational arithmetic LU factorization approach, as demonstrated by several computational tests. This is significant based on the pivotal role that this less efficient approach continues to occupy in exact mathematical programming. Indeed, as this dissertation explains, the ultimate goal of this research is to replace the in-use exact rational arithmetic LU factorization approach for validating basic solutions with the tools herein developed. We remark, however, that the featured approach is not intended to be used in every situation. The REF factorization framework is comparatively most effective when solving numerically difficult and more intricate problems, that is, those whose solution coefficients have larger bit-lengths. Conversely, as the literature review discusses, there are other techniques that are better suited for solving simpler problems, which are characterized as those whose solution coefficients have small bit-lengths. Be that as it may, it is generally not possible to determine a problem's solution size a priori and, consequently, the REF factorization framework is a more suitable general-purpose basic solution validation tool.

This work also introduces efficient algorithms for updating the REF LU and Cholesky factorizations. This further direction is necessary because applying the traditional delete-insert-reduce approach to update the REF factorizations turns out to be inefficient in terms of matrix entry growth and increased computational effort.

In fact, this inefficiency virtually wipes out all the computational savings expected of the factorization update process. Hence, the current work develops REF update algorithms that differ significantly from their traditional counterparts. The featured updates are column addition, deletion, and replacement with respect to the REF LU factorization. Additionally, the current work proves that the corresponding row updates can be performed via the REF column updates and it discusses special considerations for updating the REF Cholesky factorization.

REFERENCES

[1] John Abbott and Thorn Mulders. How tight is Hadamard's bound? *Experimental Mathematics*, 10(3):331–336, 2001.

[2] Howard Anton. *Elementary Linear Algebra*. Hoboken, NJ, USA: John Wiley & Sons, 2010.

[3] David L Applegate, William Cook, and Sanjeeb Dash. QSopt LP solver. Available at www.math.uwaterloo.ca/∼bico//qsopt, 2011.

[4] David L Applegate, William Cook, Sanjeeb Dash, and Daniel G Espinoza. Exact solutions to linear programming problems. *Operations Research Letters*, 35(6):693–699, 2007.

[5] David L Applegate, Sanjeeb Dash, William Cook, and Daniel G Espinoza. QSopt_ex rational LP solver. Available at http://www.dii.uchile.cl/∼daespino, 2007.

[6] Douglas N Arnold. The patriot missile failure. Available at https://www.ima.umn.edu/∼arnold/disasters/patriot.html, 2000.

[7] David-Olivier Azulay and Jean-Francois Pique. A revised simplex method with integer Q-matrices. *ACM Transactions on Mathematical Software*, 27(3):350–360, 2001.

[8] Erwin H Bareiss. Sylvester's identity and multistep integer-preserving Gaussian elimination. *Mathematics of Computation*, 22(103):565–578, 1968.

[9] Erwin H Bareiss. Computational solutions of matrix problems over an integral domain. *IMA Journal of Applied Mathematics*, 10(1):68–104, 1972.

[10] Richard H Bartels and Gene H Golub. The simplex method of linear programming using LU decomposition. *Communications of the ACM*, 12(5):266–268, 1969.

[11] Mokhtar S Bazaraa, John J Jarvis, and Hanif D Sherali. *Linear Programming and Network Flows*. Hoboken, NJ, USA: John Wiley & Sons, 2010.

[12] Commandant Benoît. Sur la méthode de résolution des équations normales, etc. (procédés du commandant Cholesky). *Bulletin Géodésique*, 2:67–77, 1924.

[13] Elwyn R Berlekamp. Factoring polynomials over finite fields. *Bell System Technical Journal*, 46(8):1853–1859, 1967.

[14] Liang Chen and Michael Monagan. Algorithms for solving linear systems over cyclotomic fields. *Journal of Symbolic Computation*, 45(9):902–917, 2010.

[15] Zhuliang Chen and Arne Storjohann. A BLAS based C library for exact linear algebra on integer matrices. In *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, pages 92–99. New York, NY, USA: Association for Computing Machinery, 2005.

[16] George E Collins and Mark J Encarnación. Efficient rational number reconstruction. *Journal of Symbolic Computation*, 20(3):287–297, 1995.

[17] William Cook and Daniel E Steffy. Solving very sparse rational systems of equations. *ACM Transactions on Mathematical Software*, 37(4):39, 2011.

[18] George B Dantzig. Maximization of a linear function of variables subject to linear inequalities. In *Activity Analysis of Production and Allocation,* Cowles Commission Monograph 13 (T. C. Koopmans, editor). New York, NY, USA: Wiley, 1951.

[19] Patriot Missile Defense. Software problem led to system failure at Dhahran, Saudi Arabia. *U.S. Governement Accountability Office Reports, Report Number GAO/IMTEC-92-26*, 1992.

[20] Marcel Dhiflaoui, Stefan Funke, Carsten Kwappik, Kurt Mehlhorn, Michael Seel, Elmar Schömer, Ralph Schulte, and Dennis 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*, pages 255–256. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[21] John D Dixon. Exact solution of linear equations using P-adic expansions. *Numerische Mathematik*, 40(1):137–141, 1982.

[22] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Transactions on Mathematical Software*, 35(3):19, 2008.

[23] Wayne Eberly, Mark Giesbrecht, Pascal Giorgi, Arne Storjohann, and Gilles Villard. Solving sparse rational linear systems. In *Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation*, pages 63–70. New York, NY, USA: Association for Computing Machinery, 2006.

[24] Jack Edmonds. Systems of distinct representatives and linear algebra. *Journal of Research of the National Bureau of Standards, Section B*, 71:241–245, 1967.

[25] Jack Edmonds and J-F Maurras. Note sur les q-matrices d'Edmonds. *RAIRO. Recherche Opérationnelle*, 31(2):203–209, 1997.

[26] Joseph M Elble and Nikolaos V Sahinidis. A review of the LU update in the simplex algorithm. *International Journal of Mathematics in Operational Research*,

4(4):366–399, 2012.

[27] Adolfo R Escobedo and Erick Moreno-Centeno. Roundoff-error-free algorithms for solving linear systems via Cholesky and LU factorizations. *INFORMS Journal on Computing*, 27(4):677–689, 2015.

[28] Daniel G Espinoza. *On Linear Programming, Integer Programming and Cutting Planes*. PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA, 2006.

[29] Daniel G Espinoza and Marcos Goycoolea. EGlib, Efficient General Library. Available at http://dii.uchile.cl/∼daespino/EGlib_doc, 2003.

[30] Xin G Fang and George Havas. On the worst-case complexity of integer Gaussian elimination. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 28–31. New York, NY, USA: Association for Computing Machinery, 1997.

[31] Anthony V Fiacco and Garth P McCormick. The sequential unconstrained minimization technique (SUMT) without parameters. *Operations Research*, 15(5):820–827, 1967.

[32] John JH Forrest and John A Tomlin. Updated triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical Programming*, 2(1):263–278, 1972.

[33] George E Forsythe and Cleve B Moler. *Computer Solution of Linear Algebraic Systems*, volume 7. Englewood Cliffs, NJ, USA: Prentice-Hall, 1967.

[34] Bernd Gärtner. Exact arithmetic at low cost–a case study in linear programming. *Computational Geometry*, 13(2):121–139, 1999.

[35] W Morven Gentleman and Stephen C Johnson. Analysis of algorithms, a case study: Determinants of matrices with polynomial entries. *ACM Transactions on Mathematical Software*, 2(3):232–241, 1976.

[36] Philip E Gill, Walter Murray, Michael A Saunders, and Matgaret H Wright. Maintaining LU factors of a general sparse matrix. *Linear Algebra and its Applications*, 88:239–270, 1987.

[37] Ambros M Gleixner. *Exact and Fast Algorithms for Mixed-Integer Nonlinear Programming*. PhD thesis, Technische Universität Berlin, 2015.

[38] Ambros M Gleixner, Daniel E Steffy, and Kati Wolter. Improving the accuracy of linear programming solvers with iterative refinement. In *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, pages 187–194. New York, NY, USA: Association for Computing Machinery, 2012.

[39] Ambros M Gleixner, Daniel E Steffy, and Kati Wolter. Iterative refinement for linear programming. *INFORMS Journal on Computing*, 28(3):449–464, 2016.

[40] Gene H Golub and Charles F Van Loan. *Matrix Computations*, volume 3. Baltimore, MD, USA: JHU Press, 2012.

[41] Jacek Gondzio. Interior point methods 25 years later. *European Journal of Operational Research*, 218(3):587–601, 2012.

[42] Torbjörn Granlund and the GMP development team. GNU MP: The GNU multiple precision arithmetic library. Available at http://gmplib.org, 2012.

[43] Jacques Hadamard. Résolution d'une question relative aux déterminants. *Bulletin des Sciences Mathématiques*, 17(1):240–246, 1893.

[44] Tudor Jebelean. Practical integer division with karatsuba complexity. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Com-*

putation, pages 339–341. New York, NY, USA: Association for Computing Machinery, 1997.

[45] Erich Kaltofen. An output-sensitive variant of the baby steps/giant steps determinant algorithm. In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, pages 138–144. New York, NY, USA: Association for Computing Machinery, 2002.

[46] Erich Kaltofen and Austin Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica*, 24(3-4):331–348, 1999.

[47] Erich Kaltofen and B David Saunders. On Wiedemann's method of solving sparse linear systems. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 29–38. New York, NY, USA: Springer-Verlag, 1991.

[48] Victor Klee and George J Minty. How good is the simplex algorithm. Technical report, Defense Technical Information Center, Department of Mathematics, University of Washington, Seattle, WA, 1970.

[49] Ed Klotz. Identification, assessment, and correction of ill-conditioning and numerical instability in linear and integer programs. *TutORials in Operations Research: Bridging Data and Decisions*, pages 54–108, 2014.

[50] Donald E Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Professional, 2nd edition, 1981.

[51] Thorsten Koch. The final NETLIB-LP results. *Operations Research Letters*, 32(2):138–142, 2004.

[52] Lam Lay-Yong and Shen Kangshen. Methods of solving linear equations in traditional China. *Historia Mathematica*, 16(2):107–122, 1989.

[53] Hong R Lee and B David Saunders. Fraction free Gaussian elimination for sparse matrices. *Journal of Symbolic Computation*, 19(5):393–402, 1995.

[54] Derrick H Lehmer. Euclid's algorithm for large numbers. *American Mathematical Monthly*, pages 227–233, 1938.

[55] Jacques-Louis Lions et al. Ariane 5 flight 501 failure. Technical report, European Space Agency, July 1996. Available at http://www.di.unito.it/∼damiani/ariane5rep.html, 1996.

[56] François Margot. Testing cut generators for mixed-integer linear programming. *Mathematical Programming Computation*, 1(1):69–95, 2009.

[57] James L Massey. Shift-register synthesis and BCH decoding. *Information Theory, IEEE Transactions on*, 15(1):122–127, 1969.

[58] Hans D Mittelmann. Benchmarks for optimization software. Available at http://plato.la.asu.edu/bench.html, 2002.

[59] Hans D Mittelmann. LPtestset. Available at http://plato.asu.edu/ftp/lptestset, 2006.

[60] Niels Möller. On schönhages algorithm and subquadratic integer GCD computation. *Mathematics of Computation*, 77(261):589–607, 2008.

[61] René M Montante-Pardo and Marco A Méndez-Cavazos. Un método númerico para cálculo matricial. *Revista Técnico-Científica de Divulgación de la Facultad de Ingeniería Mecánica y Eléctrica de la Universidad Autónoma de Nuevo León*, 2:1–24, September 1977.

[62] Thom Mulders and Arne Storjohann. Rational solutions of singular linear systems. In *Proceedings of the 2000 International Symposium on Symbolic and*

Algebraic Computation, pages 242–249. New York, NY, USA: Association for Computing Machinery, 2000.

[63] George C Nakos, Peter R Turner, and Robert M Williams. Fraction-free algorithms for linear and polynomial equations. *ACM SIGSAM Bulletin*, 31(3):11–19, 1997.

[64] Arnold Neumaier and Oleg Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Mathematical Programming*, 99(2):283–296, 2004.

[65] Jorge Nocedal and Stephen Wright. *Numerical Optimization*. New York, NY, USA: Springer Science & Business Media, 2006.

[66] William H Press. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge, United Kingdom: Cambridge University Press, 2007.

[67] Enrique S Quintana-Ortí and Robert A van de Geijn. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software*, 35(2):11, 2008.

[68] John K Reid. A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases. *Mathematical Programming*, 24(1):55–69, 1982.

[69] Michael A Saunders. Large-scale linear programming using the Cholesky factorization. Technical report, Stanford University Department of Computer Science, California, 1972.

[70] Arnold Schönhage. Schnelle berechnung von kettenbruchentwicklungen. *Acta Informatica*, 1(2):139–144, 1971.

[71] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4):281–292, 1971.

[72] Alexander Schrijver. *Theory of Linear and Integer Programming*. Hoboken, NJ, USA: John Wiley & Sons, 1998.

[73] Victor Shoup. NTL: A library for doing number theory. Available at http://www.shoup.net/ntl, 2008.

[74] Daniel E Steffy. Exact solutions to linear systems of equations using output sensitive lifting. *ACM Communications in Computer Algebra*, 44(3/4):160–182, 2011.

[75] Daniel E Steffy. *Topics in Exact Precision Mathematical Programming*. PhD thesis, Algorithms, Combinatorics and Optimization, Georgia Institute of Technology, Atlanta, GA, 2011.

[76] Josef Stein. Computational problems associated with racah algebra. *Journal of Computational Physics*, 1(3):397–405, 1967.

[77] John Stillwell. *Mathematics and its History*. New York, NY, USA: Springer-Verlag, 2010.

[78] Gilbert Strang. *Linear Algebra and Its Applications*. Cambridge MA, USA: Academic Press, 1976.

[79] Leena M Suhl and Uwe H Suhl. A fast LU update for linear programming. *Annals of Operations Research*, 43(1):33–47, 1993.

[80] Uwe H Suhl and Leena M Suhl. Computing sparse LU factorizations for large-scale linear programming bases. *ORSA Journal on Computing*, 2(4):325–335, 1990.

[81] Alan M Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[82] Silvio Ursic and Cyro Patarra. Exact solution of systems of linear equations with iterative methods. *SIAM Journal on Algebraic Discrete Methods*, 4(1):111–115, 1983.

[83] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge, United Kingdom: Cambridge University Press, 2013.

[84] Zhendong Wan. An algorithm to solve integer linear systems exactly using numerical methods. *Journal of Symbolic Computation*, 41(6):621–632, 2006.

[85] Debora Weber-Wulff. Rounding error changes parliament makeup. *The Risks Digest*, 13(37), 1992.

[86] Douglas Wiedemann. Solving sparse linear equations over finite fields. *Information Theory, IEEE Transactions on*, 32(1):54–62, 1986.

[87] James Hardy Wilkinson. *Rounding Errors in Algebraic Processes*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1963.

[88] Roland Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.

[89] Ronald Wunderling. SoPlex: the sequential object-oriented simplex class library. Available at http://soplex.zib.de, 1997.

[90] Wenqin Zhou and David J Jeffrey. Fraction-free matrix factors: new forms for LU and QR factors. *Frontiers of Computer Science in China*, 2(1):67–80, 2008.