


Automatic Verification of Right-greedy Numerical Linear Algebra Algorithms

Carl Kwan 

The University of Texas at Austin
Austin, TX, United States of America
carlkwan@cs.utexas.edu

Warren A. Hunt, Jr. 

The University of Texas at Austin
Austin, TX, United States of America
hunt@cs.utexas.edu

Abstract—We present an automatic verification process for formally proving the correctness of a class of greedy numerical linear algebra algorithms. To demonstrate our methodology, we present theorem prover verifications of LU and Cholesky decompositions. We formalize a framework for reasoning about matrices and matrix algorithms by partitioning matrices and developing a generalized form of the inductive invariant common to this class of greedy algorithms. Our framework enables users to readily verify any algorithm in this class automatically by simply defining the algorithm itself and specifying the class of matrices on which the algorithm performs. Our framework is also adaptable to other greedy numerical linear algebra algorithms. To our knowledge, this is the first automatic approach to verifying an entire class of numerical linear algebra algorithms.

Index Terms—Numerical linear algebra, LU decomposition, Cholesky factorization theorem, Automated theorem proving.

I. INTRODUCTION

Linear algebra is everywhere, permeating across the natural, mathematical, social, applied, and, in particular, computing sciences. The prevalence of linear algebra includes critical applications in which linear algebra computations are used to build modern infrastructure, perform data analysis affecting policy making, engineer control systems, secure information, create scientific models, and develop hardware, software, and cyberphysical systems. Determining the correctness of these linear algebra computations is vital. Numerical linear algebra concerns algorithms designed to perform such computations accurately. However, implementations of such algorithms can still fail with disastrous consequences. The potential human and capital losses due to inadequate numerical implementations necessitates formal methods.

We present a formal method for automatically verifying a class of greedy numerical linear algebra algorithms. We demonstrate the utility of our approach by verifying a particular flavor of the LU and Cholesky decompositions; that is, we verify that the product of the decomposed matrix is the original matrix itself under the appropriate conditions. We choose these specific decompositions because the development or improvement of any new family of numerical linear algebra algorithms typically begins with one of the “three amigos”: LU, Cholesky, or QR decomposition. This makes LU and Cholesky two of the most ubiquitous algorithms in numerical

methods. In this paper, we describe our approach to their mechanization.

To the best of our knowledge, any verification of numerical linear algebra algorithms by way of theorem prover is a new area of research. By applying our approach to two of the three amigos, we intend to embark on a significant line of research involving the systematic and portable verification of *families* of numerical linear algebra algorithms. One major novel contribution we make is to identify the level of abstraction appropriate for reasoning with theorem provers. If we implement matrix algorithms and reason at the level of their scalar entries, such as in Algorithm 1, then our proofs would be intractable because the mathematical expressions quickly become too large and unwieldy. Moreover, such an approach is not easily portable to other algorithms, even if they are in the same family. If we reason at too high a level, then verifying instantiated algorithms would require significant user effort, thus reducing automation. In this paper, we apply a partitioned approach to reasoning about matrices and their algorithms, which enable our verification of LU and Cholesky decompositions using the same shared framework.

LU and Cholesky decompositions are vital numerical techniques with broad applications in linear algebra. LU decomposition factorizes a matrix into a product of lower and upper triangular matrices. Cholesky decomposition specifically applies to symmetric positive definite matrices, breaking them down into the product of a lower triangular matrix and its transpose. Let

- $LU(A)$ compute the LU decomposition of A ;
- $\text{Chol}(A)$ compute Cholesky decompositions of A ;
- L_u be the strictly lower triangular part of $LU(A)$, placing 1s on the diagonal;
- U retrieve the upper triangular part of $LU(A)$.
- L be the lower triangular part of $\text{Chol}(A)$; and

Specifically, we verify that

- 1) if every principal leading submatrices of A is nonsingular, then $A = L_u U$; and
- 2) if A is symmetric positive definite, then $A = LL^T$.

We discuss the conditions on A later. Both LU and Cholesky decompositions facilitate solving linear systems, performing matrix inversions, and calculating determinants efficiently. Cholesky is particularly useful when solving linear least-

This work was supported in part by Intel Corporation and Amazon Science.

squares problems, performing Monte Carlo simulations, and optimizing quadratic forms. These sorts of decompositions play crucial roles in numerical stability, computational efficiency, and analysing the accuracy of solutions, making them fundamental tools in diverse fields such as physics, engineering, finance, and machine learning.

The particular variants of LU and Cholesky decomposition in this paper are sometimes known as “right-greedy”. “Right-greedy” refers to the particular submatrices that are updated in a partitioned representation of the matrix on which an algorithm operates during the main body of a loop or recursion. At each step of the algorithm, we update the nearest submatrices into the desired form. As the algorithm proceeds, the components are updated from left to right¹. Our method enables the theorem prover verification of a right-greedy algorithm automatically with very few user-provided hints. To discharge the algorithmic proof of correctness using a theorem prover, the only knowledge necessary is the matrix partitioning and a recognizer for the class of matrices on which the algorithm is expected to operate. Our approach is the first to enable the mechanical verification of an entire family of numerical linear algebra algorithms.

We perform our modeling and verification with ACL2, an industrial-strength first-order logic automated theorem prover with support for the real numbers via non-standard analysis [1], [2]. One advantage of using ACL2 is that the structure of the numerical linear algebra algorithms in which we are interested is well suited to ACL2’s extensive support for rewriting and induction. Another advantage of using ACL2 is the support for execution via an underlying Lisp interpreter defined within the theorem prover logic. ACL2 is unique among theorem provers in its capability execute code at the speeds of modern programming languages within the theorem prover itself, making our verified numerical linear algebra programs directly applicable to real-world problems. We use the ACL2 language to model commercial linear algebra applications, and we analyze such codes mechanically for their fitness to purpose using the ACL2 theorem prover.

There are very few verification efforts for numerical linear algebra. First, the sheer number of numerical algorithms, even for linear algebra, is daunting, and new ad hoc algorithms for specific applications are often being published. Verifying just the algorithms for critical applications would be an endless affair, requiring large-scale organization and resources. Our work addresses the verification of not just one algorithm or application, but an entire *family* of numerical algorithms. Second, different algorithms can have different structures and correctness criteria. This suggests separate proofs for each individual algorithm. Even identifying structures that are exploitable for verification purposes is challenging. Third, there is an over reliance of indexing in typical presentations of numerical algorithms. Consider the LU decomposition algorithm in Algorithm 1, which is representative of numerical algorithms [3]. Here,

¹Section IV provides a visual treatment of “right-greedy”.

Algorithm 1 Right-greedy LU decomposition (Stewart). [3]

```

for  $k = 1 : n - 1$  do
  if  $A[k, k] = 0$  Error
   $A[k + 1 : n, k] = A[k + 1 : n, k] / A[k, k]$ 
   $A[k + 1 : n, k + 1 : n]$ 
     $= A[k + 1 : n, k + 1 : n] - A[k + 1 : n, k] A[k, k + 1 : n]$ 

```

- $A[a, b]$ refers to the scalar in the a -th row and b -th column,
- $A[a : b, c]$ refers to the column vector from row a to row b in column c ,
- $A[a, b : c]$ refers to the row vector from column b to column c in row a ,
- $A[a : b, c : d]$ refers to the submatrix from row a to row b and from column c to column d .

Indexing obfuscates the design and intent of the algorithm to the point where it is unclear what is a matrix, what is a vector, or even what is a scalar in the main loop. Unbounded proofs for algorithms of this sort can be discharged by induction or rewriting by a general purpose theorem prover. Fourth, ACL2 has extensive support for execution within its logic and numerical linear algebra algorithms are usually designed to be executed. Some theorem provers can generate executable source or machine code but these tend to be unverified and unoptimized, which limits their utility especially since unverified but optimized numerical linear algebra libraries already exists. Fifth, the scope of linear algebra algorithms is sometimes limited to a class of matrices for which the definition is not conducive to formalization. For example, Cholesky decomposition is designed for matrices that are symmetric positive definite, and the usual definition for positive definite involves quantifying over all vectors. We want to avoid quantifiers in the definition of, say, positive definite because they limit the execution of a recognizer for such matrices.

In this paper we provide our solutions to the five challenges described. To address the first two challenges and partially address the third, we take advantage of the Formal Linear Algebra Methods Environment (FLAME) [4]. FLAME is an approach for systematically deriving numerical linear algebra algorithms that circumvents the problem of indexing by representing numerical linear algebra algorithms in terms of operations on the submatrices in a partitioned form of the original matrix. The partitioned form is not only more readable from a human perspective, but also exposes invariants that facilitates ACL2 reasoning and verification. The problem with algorithms in the original FLAME approach is that they are loop-based and mathematical correctness follows from identifying loop-invariants. In our approach, we recast loops into recursions and instead identify generalizable *inductive* invariants, which better aligns with the spirit of ACL2.

To address the last two challenges and finish addressing the third, we develop ACL2 mechanisms to enable automatic reasoning about linear algebra algorithms, define constructive recognizers for the matrices on which these algorithms operate,

and execute them. It is important for these recognizers to be executable because they can also serve as an efficient way to check whether a matrix is part of a solvable problem before performing more costly procedures. Execution is handled natively by ACL2. To reason about matrix algorithms, we develop our own ACL2 rules for partitioning matrices and finding inductive invariants. Another contribution we make is to identify and develop definitions that enable constructive recognizers for matrices that satisfy an algorithm’s precondition.

The rest of this paper is organized as follows: first, we discuss the limited existing literature on linear algebra and theorem proving; second, we introduce the basics of ACL2 and linear algebra; third, we motivate our mechanical method for automatically verifying numerical linear algebra algorithms by describing how to verify LU and Cholesky decompositions; fourth, we describe how to generalize the techniques used to verify our two motivating examples; finally, we conclude by summarizing our approach and discussing its immediate application and future work.

II. RELATED WORK

While theories of matrices and vector space exists in ACL2 and other theorem provers, there are practically no theorem prover verifications of numerical linear algebra algorithms. For the ACL2 theorem prover, the closest relevant existing paper of which we are aware is a formalization and proof of correctness for LU decomposition [5]. There has also been ACL2 work on using abstract single threaded objects to compute the column echelon form of a matrix [6]. Other relevant ACL2 papers include formalizations of matrices [7], [8], vectors (both real and abstract) [9], [10], and vector-valued functions [11]. An application of ACL2 matrices is the VWSIM circuit simulator for rapid, single-flux, quantum (RSFQ) circuits, which is written in ACL2 and based on repeatedly solving linear systems of the form $Ax = b$ [12]. However, VWSIM’s matrix solver is not ACL2 verified.

Expanding the purview to theorem provers in general, we find formal theories for matrices that either do not support execution or are limited to basic matrix arithmetic operations (e.g., addition, multiplication, etc.). These include Coq, Lean, Isabelle, and HOL4. In the Coq community, there are at least five proposed formal models for basic matrix theory and recent work towards integrating them has been published [13]. There is a Lean proof that positive definite matrices have an LDL decomposition [14]; however, none of the functions involved in the proof are computable. Isabelle formalizations of many matrix procedures, including algorithms for Gauss-Jordan elimination, Schur decomposition, and finding various normal forms, are in the Archive of Formal Proofs but none are natively executable [15]. There is also a HOL4 theory for basic matrix ideas and operations [16]. While many of these theorem provers are excellent at modelling mathematics, they have little to no support for the direct execution of numerical code, making them unsuited to our purposes.

FLAME is a major influence on our work. In addition to describing how to derive families of numerical linear

algebra algorithms and demonstrate their correctness based on different loop-invariants, FLAME also provides an alternate partitioning-based framework for backwards error analysis [17]. However, no formal method tools are used in the FLAME project and FLAME algorithms are not recursive. While FLAME derivations of algorithms such as LU and Cholesky decompositions indicate a natural inductive step, its mathematical proofs for the correctness and existence of these decompositions deviate significantly from our approach. Our approach to correctness is to define a recursive variant of the algorithms of interest and then constructively define a recognizer which induces an induction on the partitioning of the matrix. Existence follows because we posit an explicit algorithm which computes the desired decomposition.

No prior theorem-prover-based work supports FLAME-style analyses. ACL2-specified algorithms are our best option. Our work is the first to provide techniques for formally verifying families of executable numerical linear algebra algorithms.

III. ACL2 BASICS

Our theorem prover of choice is ACL2, a first-order logic with support for highly automated reasoning by way of extensive rewriting heuristics and induction. ACL2 contains many built-in features and tooling which support software engineering efforts, proof and theory management, user-controlled rewriting, file I/O and parsing of large-scale designs, calling internal and external automated solvers in a sound manner, and much more, all with extensive publicly-available documentation. ACL2 formalizes an applicative subset of pure Common Lisp, which enables ACL2 code to be efficiently compiled and executed.

In ACL2, functions are total, that is, all functions map all objects in the logic. By first-order, we mean quantifiers can only predicate over individuals (though we avoid explicit quantifiers in practice). The return on this restriction is that first-order logic theorem proving is highly developed, semi-decidable, and allows for truly automated reasoning. ACL2 is primarily based on term-rewriting, which is a set of rules for replacing one logical expression with another equivalent expression. Sophisticated heuristics for rewriting and extensive support for automatic induction allows ACL2 to be a highly automated and efficient tool appealing to commercial applications. ACL2 is sometimes referred to as an industrial-strength theorem prover, where it ensures the correctness of critical systems. ACL2 has been deployed to verify industrial-scale hardware designs and software systems at companies such as Intel, AMD, Oracle, Collins Aerospace, IBM, and ARM [18].

Table 1 lists some commonly used ACL2 functions, macros, and commands. A comprehensive description of the built-in ACL2 functions can be found in the ACL2 documentation [19]. Table 2 lists some commonly used ACL2 linear algebra functions which we do not further describe in this paper. We take advantage of some defined primitive matrix functions [7], but define our own functions to support reasoning about numerical linear algebra algorithms, accessing their results, and executing the algorithms themselves. For

functions which are central to this paper, such as recognizers for nonsingular matrices, more implementation details will be provided as we discuss the verification process.

Finally, we make a distinction between vanilla ACL2 and ACL2(r). Vanilla ACL2 numbers only include rationals and complex rationals. ACL2(r) is the version of ACL2 with support for real and complex numbers in general via non-standard analysis. In either case, computations on concrete numerics (rational, floating-point, or otherwise) are handled by the Common Lisp backend of ACL2 / ACL2(r), which enables the theorem prover to support native execution at modern speeds. In this paper, ACL2(r) is only necessary for taking square roots in the Cholesky decomposition algorithm. The square-root function used is the logical definition `acl2-sqrt`, which involves operations on nonstandard numbers. To make execution more amenable, we deploy a version of Cholesky which employs an iterative square-root function `sqrt-iter`, which has been verified to converge to `acl2-sqrt` [2], as a drop in replacement. It is possible to reason about square roots in vanilla ACL2 using only its algebraic properties, e.g., by augmenting the field of ACL2 numbers with some $\sqrt{}$. Instead of developing a new theory in ACL2, we decided to simply use ACL2(r).

IV. LINEAR ALGEBRA BASICS

One core idea of our approach is to recast algorithms in terms of operations on submatrices in a partitioned form of the original matrix. Originally, this partitioning was meant to make linear algebra code more intelligible and reasoning from a human perspective easier. However, it also enables machine reasoning in a verification context, which we will discuss in Sections V and VI. To see this partitioning in action, we derive the LU decomposition. An LU decomposition for a matrix A are matrices L and U where L is lower triangular with “1”s on the diagonal (i.e. unit lower triangular), U is upper triangular, and $A = LU$. In the interest of memory optimization, the unit lower triangular requirement makes it possible to overwrite the upper part of A with the upper part of U and the strictly lower part of A with the strictly lower part of L during the algorithm. Partition A , L , and U as follows:

$$A := \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}, \quad L := \begin{pmatrix} 1 & \\ \ell_{21} & L_{22} \end{pmatrix}, \\ U := \begin{pmatrix} v_{11} & u_{12}^T \\ & U_{22} \end{pmatrix}.$$

Before we continue, a note on notation: lower-case Greek letters are field scalars; lower-case Latin letters are vectors; upper-case Latin letters are matrices; and assume that any posed variables are “conformal”, e.g., if A is $m \times n$, then a_{21} is $(m-1) \times 1$ and a_{12}^T is $1 \times (n-1)$. Setting $A = LU$ gives

$$\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} 1 & \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} v_{11} & u_{12}^T \\ & U_{22} \end{pmatrix}. \quad (1)$$

We want Equation (1) to hold after performing the algorithm, i.e.

$$\alpha_{11} = v_{11}, \quad a_{21} = v_{11}\ell_{21}, \\ a_{12}^T = u_{12}^T, \quad A_{22} = \ell_{21}u_{21}^T + L_{22}U_{22}.$$

Since A is given, u_{12}^T and v_{11} are obvious. Solving for the remaining components of L and U forces

$$\ell_{21} = a_{21}\alpha_{11}^{-1},$$

and

$$L_{22}U_{22} = A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^T. \quad (2)$$

This suggests an algorithm which requires merely updating a_{21} and A_{22} . In particular, Equation (2) in the derivation above suggests a natural induction hypothesis which facilitates a recursive algorithm, i.e. our recursive call is to simply call the same LU decomposition algorithm on the smaller matrix $A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^T$.

Consider our version of LU decomposition in Algorithm 2. Contrasting Algorithm 2 with Algorithm 1 elucidates the advantages of viewing numerical linear algebra algorithms through the lens of partitioned matrices. In terms of aesthetics alone, Algorithm 2 is more elegant than Algorithm 1. The technical advantages of this is that coherent code facilitates intelligent modularity, software reliability, codebase maintenance, and high performance, while enhancing confidence in its correctness.

Indeed, for our purposes, the major advantage of the presentation in Algorithm 2 is that partitioning the matrix at the start and end of the algorithm exposes an inductive invariant. At the start of the algorithm, all components of the matrix are highlighted red, indicating that none of the present components are in the desired “LU” form. The inductive invariant is that by the time a recursive call is initiated, all but the “bottom right” component is green, indicating that everything except A_{22} is already in LU form. To remedy the final form, Equation (2) indicates that we should simply call LU on A_{22} .

The progress of a right-greedy algorithm is visualized in Figure 1. Step (1) represents a matrix prior to the updates in a particular recursive call. Green indicates portions of the matrix that are already in the desired form and red indicates portions of the matrix that still need to be updated. Step (2) represents the matrix while updates are made during a recursive call. Purple indicates the portions of the matrix that are being updated. Step (3) represents the matrix just prior to the next recursive call. As the algorithm progresses, the portion of the matrix not yet in in the desired form reduces in size, until no part of the matrix needs to be updated, at which point the algorithm terminates.

What makes Algorithm 2 “right-greedy” is that the four bottom right purple-colored components as shown in Step (2) of Figure 1 are the submatrices of A to be updated within a recursive call.

The visualization of Figure 1 is algorithm agnostic in that the same progression holds for any right-greedy algorithm – not just LU. Indeed, we can undergo a similar derivation

Table 1 Common ACL2 functions, macros, and other commands used in this paper.

Command	Description
defun	Define a function symbol, e.g. (defun add-1 (x) (+ x 1))
define	A richer alternative to defun; enforces guard checking and more
defthm	Name and prove a theorem, e.g. (defthm <-add-1 (< x (add-1 x)))
list	Define a list, e.g. (list 1 2 3) returns (1 2 3)
car	Returns the head of a list, e.g. (car (list 1 2 3)) returns 1
cons	Construct a pair, e.g. (cons 1 (list 2)) returns (1 2)
/	Divide two numbers or return the reciprocal of a number, e.g. (/ 1 2) or (/ 2)
acl2-sqrt	Square root of an ACL2 number, e.g. (acl2-sqrt 2)
b*	Binder for local variables; often used to simplify control flow statements

Table 2 ACL2 linear algebra functions.

Function	Intended Signature	Description
matrixp	$\mathbb{R}^{n \times m} \rightarrow \{\text{t}, \text{nil}\}$	Matrix recognizer, e.g. (matrixp (list (list 1 0 0))) returns t
m-emptyp	$\mathbb{R}^{n \times m} \rightarrow \{\text{t}, \text{nil}\}$	Empty matrix recognizer, e.g. (m-emptyp nil) returns t
m-empty	$\{\} \rightarrow \mathbb{R}^{0 \times 0}$	Returns an empty matrix, e.g. (m-empty) returns nil
mzero	$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^{n \times m}$	Returns a zero matrix, e.g. (mzero 1 1) returns (list (list 0))
row-car	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^m$	Returns the first row of a matrix
col-car	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^n$	Returns the first column of a matrix
row-cdr	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{(n-1) \times m}$	Remove a matrix's first row
col-cdr	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times (m-1)}$	Remove a matrix's first column
row-cons	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{(n+1) \times m}$	Append a row to a matrix
col-cons	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times (m+1)}$	Append a column to a matrix
m+	$\mathbb{R}^{n \times m} \times \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times m}$	Matrix addition
m*	$\mathbb{R}^{n \times m} \times \mathbb{R}^{m \times \ell} \rightarrow \mathbb{R}^{n \times \ell}$	Matrix multiplication
sm*	$\mathbb{R} \times \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times m}$	Scalar-matrix multiplication
sv*	$\mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$	Scalar-vector multiplication
out-*	$\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$	Outer product multiplication
get-L	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times m}$	Get a matrix's lower triangular part
get-U	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times m}$	Get a matrix's upper triangular part

Algorithm 2 LU decomposition (ACL2).

procedure LU($A \in \mathbb{R}^{m \times n}$)

Partition $A = \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}$

▷ If $n, m > 1$, then $\alpha_{11} \in \mathbb{R}$, $a_{21} \in \mathbb{R}^{(n-1) \times 1}$,
 $a_{12}^T \in \mathbb{R}^{1 \times (m-1)}$, $A_{22} \in \mathbb{R}^{(n-1) \times (m-1)}$

if $m = 0$ or $n = 0$ **then** ▷ Edge case

return $()$ ▷ Return an empty matrix

else if $n = 1$ **then** ▷ Base case

return $\begin{pmatrix} \alpha_{11} \\ a_{21} \alpha_{11}^{-1} \end{pmatrix}$

else if $m = 1$ **then** ▷ Base case

return A

else ▷ Recursive case

$a_{21} := a_{21} \alpha_{11}^{-1}$

$A_{22} := A_{22} - a_{21} a_{12}^T$

return $\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & \text{LU}(A_{22}) \end{pmatrix}$

(1) (2) (3)

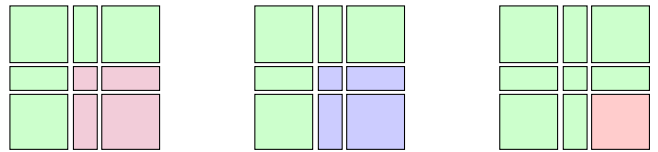


Figure 1: Progress of a right-greedy algorithm: (1) prior to updates ; (2) during updates; (3) after updates.

for the right-greedy Cholesky decomposition. Given a (real) symmetric positive definite matrix A , i.e. $A = A^T$ and $v^T A v > 0$ for all nonzero compatible vectors v , a Cholesky decomposition for A is a lower triangular matrix L such that $A = LL^T$. Partition as before:

$$A := \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}, \quad L := \begin{pmatrix} \lambda_{11} & \\ \ell_{21} & L_{22} \end{pmatrix}.$$

Setting $A = LL^T$ gives

$$\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} \lambda_{11} & \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \lambda_{11} & \ell_{21}^T \\ & L_{22}^T \end{pmatrix} \quad (3)$$

Algorithm 3 Cholesky decomposition (ACL2).

```
procedure CHOL( $A \in \mathbb{R}^{m \times n}$ )  
Partition  $A = \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}$   
▷ If  $n, m > 1$ , then  $\alpha \in \mathbb{R}$ ,  $a_{21} \in \mathbb{R}^{(n-1) \times 1}$ ,  
 $a_{12}^T \in \mathbb{R}^{1 \times (m-1)}$ ,  $A_{22} \in \mathbb{R}^{(n-1) \times (m-1)}$   
if  $m = 0$  or  $n = 0$  then ▷ Edge case  
  return  $()$  ▷ Return an empty matrix  
else if  $n = 1$  then ▷ Base case  
  return  $\begin{pmatrix} \sqrt{\alpha_{11}} \\ a_{21} \alpha_{11}^{-1} \end{pmatrix}$   
else if  $m = 1$  then ▷ Base case  
  return  $(\sqrt{\alpha_{11}} \quad a_{21}^T)$   
else ▷ Recursive case  
   $\alpha_{11} := \sqrt{\alpha_{11}}$   
   $a_{21} := a_{21} \alpha_{11}^{-1}$   
   $A_{22} := A_{22} - a_{21} a_{21}^T$   
  return  $\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & \text{CHOL}(A_{22}) \end{pmatrix}$ 
```

forces

$$\begin{aligned} \lambda_{11} &= \pm \sqrt{\alpha_{11}}, & \ell_{21} &= a_{21} \lambda_{11}^{-1}, \\ L_{22} L_{22}^T &= A_{22} - \ell_{21} \ell_{21}^T. \end{aligned} \quad (4)$$

For our purposes, we pick $\lambda_{11} = \sqrt{\alpha_{11}}$. Again, note that Equations (3) and (4) suggest a natural recursion. Our Cholesky decomposition algorithm is Algorithm 3.

Comparing Algorithm 3 with Algorithm 2 emphasizes the similar derivations, with the only contrast being the update to α_{11} in Algorithm 3. This extra update is necessary because the diagonals in a Cholesky decomposition are the same. In Algorithm 2, this update isn't necessary because we store the diagonal of an LU decomposition in L .

V. VERIFYING RIGHT-GREEDY LU DECOMPOSITION

Here we describe our verification of an LU decomposition algorithm in ACL2. There are a few points to observe in this section. First, we describe some further ACL2 details as this will be the first instance of an ACL2 program in this paper. Second, note how we specify the algorithm's conditions for success. The textbook conditions require all principal leading submatrices to be nonsingular, which is a quantified statement and undesirable for executional efficiency. LU decomposition is only one numerical linear algebra algorithm; we are interested in verifying an entire *family* of algorithms. Third, the proof of correctness for our ACL2 LU decomposition program goes hand-in-hand with the derivation in Section IV. A pen-and-paper proof may directly apply the derivation as an induction step to prove the LU decomposition correct by construction. However, in ACL2, we first *specify* the LU

decomposition algorithm as an executable program, and then prove it correct. Ideas in this section will be discussed at a higher level of abstraction in Section VII.

With the exception of some extra edge cases, Program 1 implements Algorithm 2 directly. The macro `define` is a wrapper for `defun` that simplifies many common aspects of function definition in ACL2, such as guards. Since ACL2 functions are total, guard checking is used to validate certain conditions or constraints before proceeding with execution. Guard checking is employed to enhance the robustness and reliability of ACL2 code by preventing the execution of code under inappropriate or unexpected circumstances.

The `b*` in the definition of `lu` is an example of an ACL2 macro for binding local variables with support for control flow. The first argument to `b*` is a list of “bindings” and the second argument is the ACL2 expression to which the bindings apply. For example, the binding `(alph (car (col-car A)))` declares the local variable `alph` to be equal to `(car (col-car A))`, i.e. the first element of the first column in A . If no early-exit bindings (such as `unless`) are triggered, then the value of the `b*` expression is the value of the second argument to `b*` with the bindings given by the first argument.

It is challenging to formalize the typical conditions for an LU decomposition of a matrix A to succeed. For one, they are presented as a quantified statement over the submatrices of A : all *principal leading submatrices* of A need to be nonsingular. If A is $n \times n$, the *principal leading submatrices* of A are the $k \times k$ “top left” submatrices of A , where $k \in [1, n]$. While ACL2 supports quantifiers via Skolem functions, these are not executable. We want a recognizer for LU decomposable matrices to be at least executable, not to mention efficient, because: (1) it can serve as a guard; and (2) our recognizer will also serve to induce an induction scheme for proving the correctness of `lu`. The other problem with the typical conditions is that nonsingularity is challenging to formalize. Thanks to the Invertible Matrix Theorem, there are over 20 equivalent characterizations for nonsingularity, most of which are computationally inefficient, require significant theory building, or also involve quantified statements.

Our solution is to use *Schur complements*. Consider Equation (1)

$$\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} 1 & \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} v_{11} & u_{12}^T \\ & U_{22} \end{pmatrix}. \quad (1 \text{ revisited})$$

Notice that if $\alpha_{11} \neq 0$, then A is LU decomposable iff the RHS of Equation (2)

$$L_{22} U_{22} = A_{22} - a_{21} \alpha_{11}^{-1} a_{12}^T \quad (2 \text{ revisited})$$

is also LU decomposable. This is interesting because it reduces the condition for a matrix to be LU decomposable into a condition about a smaller matrix, which is reminiscent of some “induction step”. Indeed, the RHS of Equation (2) is the *Schur complement* of α_{11} in A . While mathematical references commonly describe the conditions for a matrix to be LU decomposable in terms of the leading principal submatrices,

Program 1 ACL2 implementation of LU decomposition Algorithm 2.

```
(define lu ((A matrixp)) ...
(b* ;; BASE CASES
  ((unless (matrixp A)) (m-empty)) ;; If A not a matrix, return empty
  ((if (m-empty A) A) ;; If A empty, return A
  (alph (car (col-car A))) ;; alph := "top left" scalar in A
  ((if (zerop alph) ;; If alph zero, return a zero
    (mzero (row-count A) ;; matrix of the same dimensions
            (col-count A))) ;; as A
  ((if (m-empty (col-cdr A))) ;; If A is a column, return
    (row-cons (list alph) ;; [ 1 ] [ a1 ] = [ a1 ] = A
              (sm* (/ alph) ;; [ a2/a1 ] [ a2 ]
                  (row-cdr A)))) ;; [ ... ] [ ... ]
  ((if (m-empty (row-cdr A))) A) ;; If A is a row, return A

;; PARTITION
(a21 (col-car (row-cdr A))) ;; [ alph | a12 ] := A
(a12 (row-car (col-cdr A))) ;; [ ----- ]
(A22 (col-cdr (row-cdr A))) ;; [ a21 | A22 ]

;; UPDATE
(a21 (sv* (/ alph) a21)) ;; a21 := a21 / alph
(A22 (m+ A22 (sm* -1 (out-* a21 a12)))) ;; A22 := A22 - a21 * a12

;; RECURSE
(row-cons (row-car A) ;; [ alph | a12 ]
          (col-cons a21 (lu A22))) ... ;; [ ----- ]
                                             ;; [ a21 | LU(A22) ]
```

Program 2 ACL2 theorem for LU decomposition correctness.

```
(defthm lu-correctness
(b* ((LU (lu A))
  (L (get-unit-L LU))
  (U (get-U LU)))
  (implies (and (equal (col-count A)
                       (row-count A))
                (nonsingular-submatrices-p A))
            (equal (m* L U) A))))
```

Program 3 ACL2 theorem for Cholesky decomposition correctness.

```
(defthm chol-correctness
(b* ((L (get-L (chol A)))
  (Lt (mtrans L)))
  (implies (and (equal (mtrans A) A)
                 (positive-definite-p A)
                 (equal (col-count A)
                       (row-count A)))
            (equal (m* L Lt) A))))
```

the proof that these conditions are sufficient reduces to an induction step that depends on Equation (2) [3].

To see the connection with nonsingular principal submatrices, observe that if no zeros appear after k recursive steps, then the k -th principal leading submatrix is nonsingular because its determinant is nonzero. Instead of reasoning with determinants, which are rarely useful in numerical algorithms [20], Schur complements provide a more concise ACL2 condition for success and generalizes to other algorithms.

VI. VERIFYING RIGHT-GREEDY CHOLESKY DECOMPOSITION

Here we briefly describe our verification of the right-greedy Cholesky decomposition. Our focus will be on the commonalities with LU decomposition verification, some peculiarities in recognizing symmetric positive definite matrices, and less on ACL2 implementation details.

In order for a matrix A to have a Cholesky decomposition, it must be symmetric positive definite. Symmetric simply means $A^T = A$, but positive definite requires $v^T A v > 0$ for all nonzero compatible v . The latter is once again a quantified statement, which has all the implications discussed previously

in Section V. In order to define a executable recognizer for positive definite matrices, we once again look at Schur complements to satisfy *Sylvester's criterion* for a symmetric matrix to be positive definite. Sylvester's criterion states that a symmetric matrix is positive definite iff the principal leading submatrices are positive. The latter is equivalent to each principal leading submatrix having a positive determinant. From Section V, we saw that recursively computing Schur complements along the diagonal of A exhibits the determinants of the principal leading submatrices of A . Thus we merely need to check that each of these determinants are positive, which is how `positive-definite-p` in Program 3 is defined. The theorem for right-greedy Cholesky decomposition correctness then passes with minimal user-provided hints.

VII. GENERALIZING RULES FOR AUTOMATED VERIFICATION

Generalizing the ideas of Sections V and VI, our method to verifying the LU and Cholesky decompositions can be generalized to any right-greedy numerical linear algebra algorithms.

- 1) Define a recursive right-greedy algorithm.

- 2) Verify the derivation using the partitioned matrix approach.
- 3) Define a recursive recognizer for the appropriate class of matrices.
- 4) Induct according to a scheme automatically suggested by the recognizer.

The only steps which require human involvement is in Step 1 and 3. All that is required of a user is to define the algorithm to be verified and the class of matrices for which the algorithm computes. Step 4 is performed automatically because induction in ACL2 requires no human involvement. Step 2 is made automatic thanks to a formalized approach to deriving right-greedy algorithm. Observe that the RHS of both Equations (1) and (3) are simply instances of matrix multiplication between general partitioned matrices

$$\begin{aligned} BC &= \begin{pmatrix} \beta_{11} & b_{12}^T \\ b_{21} & B_{22} \end{pmatrix} \begin{pmatrix} \gamma_{11} & c_{12}^T \\ c_{21} & C_{22} \end{pmatrix} \\ &= \begin{pmatrix} \beta_{11}\gamma_{11} + b_{12}^T c_{21} & \beta_{11}c_{12}^T + b_{12}^T C_{22} \\ b_{21}\gamma_{11} + B_{22}c_{21} & b_{21}c_{12}^T + B_{22}C_{22} \end{pmatrix}. \end{aligned} \quad (5)$$

We formalize Equation (5) as an ACL2 rewrite rule which fires automatically when verifying the LU and Cholesky derivations. More generally, suppose we want to verify a right-greedy algorithm which computes B and C such that $BC = A$. The updates performed by a right-greedy algorithm's recursive step will be to compute β_{11} , γ_{11} , b_{21} , c_{21} , b_{12}^T , and c_{12}^T such that

$$\begin{aligned} \alpha_{11} &= \beta_{11}\gamma_{11} + b_{12}^T c_{21}, & a_{12}^T &= \beta_{11}c_{12}^T + b_{12}^T C_{22}, \\ a_{21} &= b_{21}\gamma_{11} + B_{22}c_{21} \end{aligned}$$

all hold. Then the algorithm's recursive call will be to find the decomposition

$$B_{22}C_{22} = A_{22} - b_{21}c_{12}^T. \quad (6)$$

The above identities are easily translated into ACL2 rewrite rules as an instantiation of the rewrite rule for Equation (5). Given these rewrite rules, the induction in Step 4 discharges automatically.

The LU and Cholesky decompositions we verify are instantiations of the above. Note that Equation (2)

$$L_{22}U_{22} = A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^T \quad (2 \text{ revisited})$$

is a case of Equation (6). If $\alpha_{11} \neq 0$ also holds, then $A = LU$. Similarly, Equation (4)

$$L_{22}L_{22}^T = A_{22} - \ell_{21}\ell_{21}^T \quad (4 \text{ revisited})$$

is a case of Equation (6). If $\alpha_{11} > 0$ and $a_{12} = a_{21}$ also hold, then $A = LL^T$. These rules follow directly from Equation (5) with little user-guidance in ACL2.

VIII. CONCLUSION

We demonstrated a formal method for automatically verifying right-greedy numerical linear algebra algorithms. At the heart of our approach is the partitioned matrix environment which we use to define and verify derivations of recursive right-greedy algorithms. Partitioning and defining algorithms

in this manner promotes automated reasoning and verification by introducing induction schemes. We've implemented our method using the ACL2 theorem prover. The choice of theorem prover is not vital provided that it supports induction. However, ACL2 provides two additional major benefits. First ACL2 offers a high degree of automation beyond what is possible with other theorem provers. Second, our verified formalizations are natively executable within the logic of ACL2; this provides industrial-level computational performance. This is particularly important because numerical algorithms are usually meant to be implemented and executed in real world systems. No other theorem prover offers the same level of execution performance.

Our work involved writing 1593 lines of new ACL2 code, used to introduce 262 new ACL2 events. Verifying the new ACL2 code required 6793576 prover steps, which were performed automatically. Performing these steps took 9.76 seconds and ACL2 used 1.37 GB of memory on a laptop. The interested reader may try using our code [21] to decompose their own matrices.

There are immediate applications for our work. We discussed determinants in Sections V and VI. Note that if $A = LU$ is LU decomposable, then $\det(A) = \det(L)\det(U) = \det(U)$ is simply the product of the diagonal of U . Another consequence of formalizing a right-greedy LU decomposition algorithm is that the computed U is actually the row echelon form of A . This means that `(defun ge (A) (get-U (lu A)))` is the ACL2 verified formalization of Gaussian elimination. One very important application of LU decomposition is that it can be used to solve a linear system $b = Ax$. If $A = LU$, then $b = Ax = (LU)x = L(Ux)$ indicates that one can first solve $b = Ly$ via forwards substitution and then $y = Ux$ via backwards substitution to solve $b = Ax$. Cholesky can be applied similarly. We have formalized backwards and forwards substitution in ACL2, which is beyond the scope of this paper, but this indicates we have a verified and executable method for solving systems of linear equations in ACL2.

The applications of numerical linear algebra in which safety, correctness, and accuracy are critical indicates a need for formally verified numerical linear algebra software systems. In addition to solving linear systems, we can pursue the verification of other executable numerical linear algebra algorithms. The class of right-greedy algorithms includes classical QR decomposition, which has yet to be formally verified. Proving this in ACL2 would give rise to verified executable implementations of LU, Cholesky, and QR decompositions (sometimes referred to as the "three amigos" by the scientific computing community), which could serve as the beginnings of a fully verified numerical linear algebra library.

Right-greedy algorithms are a major player in scientific and high-performance computing, with many dozens of such algorithms serving as the basis for ongoing research. Targeting improved performance on not just dense, but also sparse and block matrices across architectures such as GPUs and FPGAs place variants of right-looking algorithms in the hundreds. Other names for "right-greedy" are "right-looking", "data-

driven” or “submatrix”. Our approach can be augmented to verify other families of numerical linear algebra algorithms. In the FLAME framework, identifying different loop invariants suggests derivations of other algorithmic flavors, such as “left-greedy”, “up-greedy”, “bordered”, etc. We want to develop formal methods for automatically verifying these other families of numerical linear algorithms.

Another important FLAME idea is using the partitioned matrix approach to perform backwards error analysis. Formalizing bounds on errors and proving the convergence of iterative numerical algorithms are vital to the reliability of their implementations. This would involve notions such as matrix norms or the condition number of a matrix. ACL2 supports matrix and vector analysis by way of nonstandard analysis [11], [22] and recent developments in the ACL2 research community include a deep embedding of floating-point numbers into the ACL2 logic. This provides all the formal tools necessary to perform ACL2 backwards error analysis of numerical linear algebra algorithms and we intend to pursue these sorts of proofs.

Linear algebra underlies modern scientific computing infrastructure. It is critical real world linear algebra computations are accurate and correct. We endeavour to guarantee the veracity of these computations by developing verified numerical linear algebra libraries. Our method for automating the verification of right-greedy numerical linear algebra algorithms is foundational to achieving this objective.

ACKNOWLEDGMENTS

We thank Robert van de Geijn, Margaret Myers, and the anonymous reviewers for their helpful comments and feedback.

REFERENCES

- [1] M. Kaufmann and J. S. Moore, “ACL2 home page,” <https://cs.utexas.edu/~moore/acl2/>, 1997, accessed 2024-06-25.
- [2] R. A. Gamboa and M. Kaufmann, “Nonstandard analysis in ACL2,” *J. Autom. Reason.*, vol. 27, no. 4, p. 323–351, November 2001.
- [3] G. W. Stewart, *Matrix Algorithms Volume I: Basic Decompositions*, 1st ed., 1998.
- [4] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, “FLAME: Formal linear algebra methods environment,” *ACM Trans. Math. Softw.*, vol. 27, no. 4, pp. 422–455, December 2001.
- [5] C. Kwan, “Classical LU decomposition in ACL2,” *Electronic Proceedings in Theoretical Computer Science*, vol. 393, pp. 1–5, November 2023.
- [6] L. Lambán, F. J. Martín-Mateos, J. Rubio, and J.-L. Ruiz-Reina, “Using abstract stobjs in ACL2 to compute matrix normal forms,” in *Interactive Theorem Proving*, M. Ayala-Rincón and C. A. Muñoz, Eds. Cham: Springer International Publishing, 2017, pp. 354–370.
- [7] J. Hendrix, “Matrices in ACL2,” 2003. [Online]. Available: <https://www.cs.utexas.edu/users/moore/acl2/workshop-2003/contrib/hendrix/hendrix.pdf>
- [8] R. Gamboa, J. Cowles, and J. Van Baalen, “Using ACL2 arrays to formalize matrix algebra,” 2003. [Online]. Available: <https://www.cs.uwo.edu/~ruben/static/pdf/matalg.pdf>
- [9] C. Kwan and M. R. Greenstreet, “Real vector spaces and the Cauchy-Schwarz inequality in ACL2(r),” *Electronic Proceedings in Theoretical Computer Science*, vol. 280, pp. 111–127, October 2018.
- [10] C. Kwan, Y. Peng, and M. R. Greenstreet, “Cauchy-Schwarz in ACL2(r) abstract vector spaces,” *Electronic Proceedings in Theoretical Computer Science*, vol. 327, pp. 90–92, May 2020.
- [11] C. Kwan and M. R. Greenstreet, “Convex functions in ACL2(r),” *Electronic Proceedings in Theoretical Computer Science*, vol. 280, pp. 128–142, October 2018.
- [12] W. A. Hunt, Jr., V. Ramanathan, and J. S. Moore, “VWSIM: A circuit simulator,” in Proceedings Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, 26th–27th May 2022, ser. Electronic Proceedings in Theoretical Computer Science, R. Sumners and C. Chau, Eds., vol. 359. Open Publishing Association, 2022, pp. 61–75.
- [13] Z. Shi and G. Chen, “Integration of multiple formal matrix models in Coq,” in *Dependable Software Engineering. Theories, Tools, and Applications*, W. Dong and J.-P. Talpin, Eds. Cham: Springer Nature Switzerland, 2022, pp. 169–186.
- [14] “Lean mathlib3 documentation: LDL decomposition,” https://leanprover-community.github.io/mathlib_docs/linear_algebra/matrix/ldl.html, accessed 2023-07-13.
- [15] R. Thiemann and A. Yamada, “Matrices, Jordan normal forms, and spectral radius theory,” *Archive of Formal Proofs*, August 2015, https://isa-afp.org/entries/Jordan_Normal_Form.html, Formal proof development.
- [16] Z. Shi, Y. Zhang, Z. Liu, X. Kang, Y. Guan, J. Zhang, and X. Song, “Formalization of matrix theory in HOL4,” *Advances in Mechanical Engineering*, vol. 6, pp. 195–276, 2014.
- [17] P. Bientinesi and R. A. van de Geijn, “Goal-oriented and modular stability analysis,” *SIAM J. Matrix Anal. Appl.*, vol. 32, no. 1, p. 286–308, March 2011.
- [18] W. A. Hunt, M. Kaufmann, J. S. Moore, and A. Slobodova, “Industrial hardware and software verification with ACL2,” *Philosophical Transactions of the Royal Society of London Series A*, vol. 375, no. 2104, September 2017.
- [19] ACL2, *User manual for the ACL2 Theorem Prover and the ACL2 Community Books*, accessed 2024-07-12. [Online]. Available: <https://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html>
- [20] L. N. Trefethen and D. Bau, III, *Numerical Linear Algebra*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.
- [21] M. Kaufmann and J. S. Moore, “ACL2 system and community books,” <https://github.com/acl2/acl2>, 2014.
- [22] C. Kwan, “Towards formalized matrix analysis and algorithms,” in *International Symposium on Artificial Intelligence and Mathematics*, 2022.