Formalizing the Cholesky Factorization Theorem

Carl Kwan

The University of Texas at Austin, TX, United States of America

Warren A. Hunt Jr.

The University of Texas at Austin, TX, United States of America

Abstract

We present a formal proof of the Cholesky Factorization Theorem, a fundamental result in numerical linear algebra, by verifying formally a Cholesky decomposition algorithm in ACL2. Our mechanical proof of correctness is largely automatic for two main reasons: (1) we employ a derivation which involves partitioning the matrix to obtain the desired result; and (2) we provide an inductive invariant for the Cholesky decomposition algorithm. To formalize (1), we build support for reasoning about partitioned matrices. This is a departure from how typical numerical linear algebra algorithms are presented, i.e. via excessive indexing. To enable (2), we build a new recursive recognizer for a matrix to be Cholesky decomposable and mathematically prove that the recognizer is indeed necessary and sufficient. Guided by the recognizer, ACL2 automatically inducts and verifies the Cholesky decomposition algorithm. We also present our ACL2-based formalization of the decomposition algorithm itself, and discuss how to bridge the gap between verifying a decomposition algorithm and proving the Cholesky Factorization Theorem. To our knowledge, this is the first formalization of the Cholesky Factorization Theorem.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Mathematics of computing → Computations on matrices

Keywords and phrases Numerical linear algebra, Cholesky factorization theorem, Matrix decomposition, Automated reasoning, ACL2

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.25

Supplementary Material *Software (Source Code)*: `https://github.com/acl2/acl2/tree/master/books/projects/cholesky`, archived at `swh:1:dir:59ed119089ee943a78fe019c760fc4f75046e663`

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

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

1 Introduction

We present an ACL2-based formalization of the Cholesky Factorization Theorem. Our approach implements a Cholesky decomposition algorithm, CHOL, in the ACL2 logic, and we verify its correctness using the ACL2 theorem prover. Our formalization is built on existing ACL2 libraries and theories for basic vector and matrix operations (e.g. addition, multiplication, transpose, etc.), but embedding a Cholesky decomposition algorithm required a significant extension over existing theories. In addition to building support for a partitioned matrix environment and functions for accessing the lower triangular part of a matrix, we also had to develop alternate definitions for matrix operations (e.g. multiplication) and verify them against existing definitions.

We base our Cholesky formalization on the Formal Linear Algebra Methods Environment (FLAME) [8]. FLAME is an approach to systematically deriving numerical linear algebra algorithms and proving[1] them correct. We follow a FLAME derivation for a Cholesky

[1] FLAME is "formal" in the systematic sense, not in the theorem proving sense.

© Carl Kwan and Warren A. Hunt Jr.;
licensed under Creative Commons License CC-BY 4.0
15th International Conference on Interactive Theorem Proving (ITP 2024).
Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 25; pp. 25:1–25:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

decomposition algorithm but modify the algorithm to better suit ACL2's strength in recursion and induction. The advantage to using the FLAME approach is that it represents linear algebra algorithms in terms of operations on components of a matrix's partitioned representation. One common pitfall with how typical matrix algorithms are presented is the over-reliance on indexing. Intricate indices are a common cause of bugs in programs. Introducing indices for matrix / vector entries can also introduce numerous variables causing formal and automated processes to become intractable. Instead, a partitioned representation of a matrix abstracts away details that are irrelevant to the operations of interest. Another advantage to using FLAME's partitioned representation is that it exposes loop invariants. This enables us to more readily derive inductive invariants and verify the correctness of our Cholesky decomposition algorithm.

One departure from typical proofs of the Cholesky Factorization Theorem, including the one from FLAME, is that we develop a variant of *Sylvester's criterion*, a characterization for symmetric matrices to be positive definite, for use as a hypothesis in our main result. Sylvester's criterion states that a symmetric matrix is positive definite iff its principal leading submatrices have positive determinants. However, determinants do not readily lend themselves to numerical computation. Instead, we look at the diagonal of each principal leading submatrix and posit their positivity. The advantage of using this definition of symmetric positive definite is that it is recursive, amenable to ACL2 formalization, and helps automate the ACL2 proof of correctness for the decomposition algorithm. By correctness, we mean the following:

▶ **Theorem 1.** *Let A be a symmetric positive definite matrix. Let L be the lower triangular part of* CHOL*(A). Then $A = LL^T$.*

Theorem 1 permits us to prove the Cholesky Factorization Theorem.

▶ **Theorem 2** (Cholesky Factorization Theorem)**.** *If A is a symmetric positive definite matrix, then $A = LL^T$ for some lower triangular matrix L.*

The Cholesky Factorization Theorem states that symmetric positive definite matrices can be decomposed into the product of a lower triangular matrix and its transpose. While the two theorems are similar, their differences are enhanced when viewed through the lens of ACL2. One distinction is that Theorem 2 is a quantified statement and ACL2 support for quantifiers is limited. Propositional statements about functions, such as Theorem 1, is typically the preferred approach for reasoning in ACL2. The discussion in this paper focuses on issues such as these and our ACL2 formalization.

There are two primary advantages to our choice of ACL2.[2] First, ACL2 is highly automated with extensive support for rewriting. To discharge the proof of Theorem 1 in ACL2, 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 efforts required relatively few user-defined hints, lemmas, or events. Second, ACL2 supports the execution of its functions via an underlying Lisp interpreter defined within the theorem prover logic. Few theorem provers are capable of natively executing formalized functions. This makes verifying a *computational* algorithm such as the Cholesky decomposition in ACL2 particularly meaningful.

---

[2] Technically, we use ACL2(r), a version of ACL2 with support for real numbers via nonstandard analysis, which is only necessary for taking square roots in the Cholesky decomposition.

The Cholesky decomposition is fundamental to numerical linear algebra and scientific computing. For example, a common problem involves solving linear systems of the form $Ax = b$, where x and b are vectors of dimension compatible with A. If A is symmetric positive definite, then it has a Cholesky decomposition $A = LL^T$ and we obtain $LL^T x = Ax = b$. Finding x can be efficiently done by first solving $Ly = b$ via forwards substitution and then $L^T x = y$ via backwards substitution. Another practical application of Cholesky is in solving the linear least squares problem $\|y - B\hat{x}\|_2 = \min_{x \in \mathbb{R}^n} \|y - Bx\|_2$ for $\hat{x}$. By setting $A = B^T B$ and $b = B^T y$, we can find the solution to the linear least squares problem by solving $A\hat{x} = b$ in much the same way as before. In addition to these basic applications, Cholesky can be used to find matrix inverses, perform Monte Carlo simulations, and optimize quadratic forms. This makes Cholesky a vital tool in areas such as engineering, finance, and machine learning.

2 Related Work

To our knowledge, there is no other formal proof for the Cholesky Factorization Theorem. There are a few theorem prover formalizations of other decomposition algorithms. In ACL2, there is a verified executable implementation of an LU decomposition algorithm [13]. Lean's mathlib contains a formalization of LDL decomposition, but the matrix functions used in the LDL decomposition are not computable [15]. In Isabelle's Archive of Formal Proofs, there is also a formalization of Schur decomposition [20]. Basic matrix theories have long been formalized using theorem provers, including Coq [17], HOL4 [18], HOL Light [9] and the aforementioned ACL2 [10, 6, 14], Lean [16], and Isabelle [19]. Notably, ACL2's and Isabelle's theories of basic matrices provide executable operations.

Our work is inspired heavily by FLAME. While FLAME is "formal" in that it systematically derives numerical algorithms, no formal method or verification is involved with FLAME. The relevance of FLAME to our work is that FLAME introduces a partitioned matrix environment (PME), which enables us to approach Cholesky without the burden of indices. Our derivation is similar to FLAME's in that it begins with a PME, which naturally leads to a recursive Cholesky variant. However, FLAME's algorithm is loop based. The FLAME approach to systematically proving the correctness of its algorithms is to identify loop invariants. Since our Cholesky algorithm is recursive, we perform an analogous analysis to guide the verification of our Cholesky algorithm, but with induction invariants. Note a loop-invariant verification approach for a loop-based Cholesky algorithm may also be possible in ACL2 using ACL2's analogue of Common Lisp loops [2]. Because of ACL2's long tradition with recursion and the natural correspondence between the derivation and a recursive Cholesky algorithm, we opted to directly verify the recursive algorithm.

3 Deriving a Verifiable Cholesky Decomposition Algorithm

A core idea behind FLAME is to recast algorithms in terms of operations on components of a matrix's partitioned form. This is called a *partitioned matrix environment* (PME). PME is meant to make linear algebra code more intelligible, and enable the systematic derivation and pen-and-paper proofs of numerical linear algebra algorithms. However, PME also lends itself well to developing recursive matrix algorithms and induction proofs of their correctness. To demonstrate this, we derive a Cholesky decomposition algorithm in this section and verify it formally in Section 4.

Let lower-case Greek letters (e.g. α) denote real numbers, lower-case Latin letters (e.g. v) denote vectors, and upper-case Latin letters (e.g. A) denote matrices. Since Cholesky only deals with symmetric matrices, all matrices will be square. For ease of notation, assume any vectors and matrices in a posed expression are compatible (e.g. if $Ax = b$ and A is $n \times n$, then x and b are $n \times 1$).

Given a (real) symmetric positive definite matrix A, i.e. $A = A^T$ and $v^T Av > 0$ for all nonzero v, a Cholesky decomposition for A is a lower triangular matrix L such that $A = LL^T$. To derive the desired Cholesky decomposition algorithm, partition A and L as follows:

$$A := \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right), \qquad L := \left(\begin{array}{c|c} \lambda_{11} & \\ \hline \ell_{21} & L_{22} \end{array}\right).$$

Note that $a_{12} = a_{21}$ since A is symmetric. Moving forward, we drop the "bars" for simplicity. If $A = LL^T$, then

$$\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix} = A = LL^T = \begin{pmatrix} \lambda_{11} & \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \lambda_{11} & \ell_{21}^T \\ & L_{22}^T \end{pmatrix}. \tag{1}$$

This is equivalent to

$$\alpha_{11} = \lambda_{11}^2, \qquad a_{12}^T = \lambda_{11}\ell_{21}^T, \qquad a_{21} = \lambda_{11}\ell_{21}, \qquad A_{22} = \ell_{21}\ell_{21}^T + L_{22}L_{22}^T.$$

We want Equation (1) to hold. Since a potential algorithm which computes L is given A, we solve for the components of L:

$$\lambda_{11} = \pm\sqrt{\alpha_{11}}, \qquad \ell_{21} = a_{21}\lambda_{11}^{-1}, \qquad L_{22}L_{22}^T = A_{22} - \ell_{21}\ell_{21}^T.$$

For our purposes, we pick $\lambda_{11} = \sqrt{\alpha_{11}}$. Note that $L_{22}L_{22}^T$ is a Cholesky decomposition for $A_{22} - \ell_{21}\ell_{21}^T$. This suggests a algorithm which updates α_{11} and a_{21}, and recurses on $A_{22} - \ell_{21}\ell_{21}^T$. Indeed, Algorithm 1 computes a Cholesky decomposition. The three "if" branches handle base cases where the matrix is empty or only α_{11} and a_{21} are updated. The recursive step updates A_{22} as well. Note that the algorithm accepts non-square matrices. Even though we are only interested in the output of the algorithm under symmetric positive definite inputs, we nonetheless deal with the non-square cases in order to simplify the acceptance of our algorithm into the ACL2 logic.

Highlighted in Algorithm 1 are the components α_{11}, a_{21}, a_{12}^T, and A_{22} of A prior to and after their updates in the algorithm. Note that the only component of A that is passed to the recursive call is A_{22}, the "bottom right" part of A. This suggests the remaining components need not be updated anymore. Indeed, highlighted in red are the components of A that still need to be updated. Prior to entering the main body of the algorithm, all components still need to be updated. But after the updates to α_{11}, a_{21}, and A_{22}, the only component that still needs to be updated is A_{22}. The other components are already in Cholesky decomposition form. As the recursive algorithm progresses, "layers" of the matrix are replaced by its Cholesky decomposition.[3] Visually, this progression is represented in Figure 1. Step (1) represents a matrix prior to the updates in a recursive iteration. Green indicates portions of the matrix that are already Cholesky decomposed and red indicates portions of the matrix that still need to be updated. Step (2) represents the matrix within the main body of a recursive iteration of Algorithm 1. Purple indicates the portions of the

[3] This also means Algorithm 1 computes a Cholesky decomposition in place.

Algorithm 1 A recursive Cholesky decomposition algorithm.

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

  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

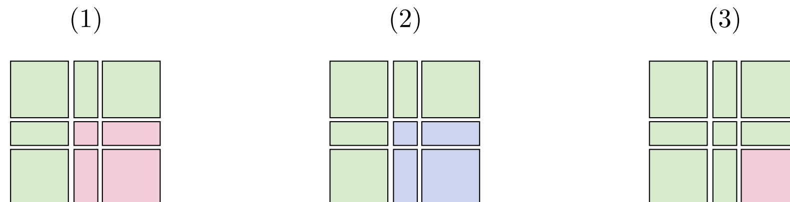
    return $\begin{pmatrix} \sqrt{\alpha}_{11} & a_{21}^T \end{pmatrix}$

  else ▷ Recursive case

    $\alpha_{11} \coloneqq \sqrt{\alpha_{11}}$
    $a_{21} \coloneqq a_{21}\alpha_{11}^{-1}$
    $A_{22} \coloneqq A_{22} - a_{21}a_{21}^T$

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

Figure 1 Progress of Algorithm 1: (1) prior to updates; (2) during updates; (3) after updates.



matrix that are being updated. Step (3) represents the matrix after the updates are made. As the algorithm progresses, the proportion of the matrix not yet Cholesky decomposed decreases, until no part of the matrix needs to be updated, at which point the algorithm terminates.

The derivation and visual progression of Algorithm 1 suggests an induction invariant, that is, performing the updates in the algorithm computes a Cholesky decomposition for all matrix components except for the "bottom right". Since the recursive call operates on a smaller matrix, the procedure eventually terminates and indeed computes a Cholesky decomposition for A. An inductive argument, with induction hypothesis stating essentially $A_{22} - \ell_{21}\ell_{21}^T = L_{22}L_{22}^T$ is Cholesky decomposable, would be sufficient to discharge a proof of the correctness of Algorithm 1, thus providing a roadmap to verifying the Cholesky Factorization Theorem formally.

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

Command	*Description*
`define`	Define a function symbol, enforce 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

4 Formally Verifying the Cholesky Factorization Theorem

To verify the Cholesky Factorization Theorem formally we need to demonstrate that every symmetric positive matrix has a Cholesky decomposition. We embed our Cholesky decomposition algorithm into the ACL2 logic, verify it, and apply it to compute a witness for the desired theorem. Table 1 lists some commonly used ACL2 functions, macros, and commands in general. Comprehensive ACL2 documentation is freely available and searchable online [3].

We employ some existing ACL2 primitive matrix functions [10] in order to formalize Algorithm 1, and we define our own functions to support reasoning about decomposition algorithms in general, accessing their results, and executing them. We also formalize alternate definitions for primitive matrix operations and prove them equivalent to the existing ones. Table 2 lists some of these ACL2 matrix functions.

4.1 Formalizing the Decomposition Algorithm

Our ACL2 formalization of Algorithm 1 is shown in Program 1. The `b*` in the definition of `chol` is 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 third binding in Program 1's `b*` is

```
(alph (car (col-car A)))
```

which declares the local variable `alph` to be equal to `(car (col-car A))`, i.e. the first element of the first column in `A`. The `b*` macro also supports early-exit bindings. For example, the first binding in the same `b*` is

```
((unless (mbt (matrixp A))) (m-empty))
```

which is triggered when `A` is not an ACL2 matrix and an empty matrix `(m-empty)` is returned. The macro `mbt` is logically equivalent to its argument (i.e. `(mbt x)` equals `x`) but immediately evaluates to `t` during runtime (ignoring `x`). This optimization is permitted by guard verification (discussed later). Provided no early exit bindings are triggered, the `b*` expression returns the second argument – in Program 1, this is

```
(row-cons (cons alph a12)
          (col-cons a21 (chol A22)))
```

Note extra edge cases, such as those handling when A is not a matrix, appear in Program 1 but not Algorithm 1. In ACL2, logical functions are total, that is, all functions map all objects

Table 2 ACL2 linear algebra functions.

Function	*Intended Signature*	*Description*
`matrixp`	$\mathbb{R}^{n\times m} \to \{\texttt{t},\texttt{nil}\}$	Matrix recognizer, e.g. `(matrixp (list (list 1 0))` returns `t`
`m-emptyp`	$\mathbb{R}^{n\times m} \to \{\texttt{t},\texttt{nil}\}$	Empty matrix recognizer, e.g. `(m-emptyp nil)` returns `t`
`m-empty`	$\{\} \to \mathbb{R}^{0\times 0}$	Returns an empty matrix, e.g. `(m-empty)` returns `nil`
`mzero`	$\mathbb{N} \times \mathbb{N} \to \mathbb{R}^{n\times m}$	Returns a zero matrix, e.g. `(mzero 1 2)` returns `((0 0))`
`row-car`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{m}$	Returns the first row of a matrix, e.g. `(row-car (list (list 1 2) (list 3 4)))` returns `(1 2)`
`col-car`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{n}$	Returns the first column of a matrix, e.g. replacing `row-car` with `col-car` in the previous example returns `(1 3)`
`row-cdr`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{(n-1)\times m}$	Remove a matrix's first row, e.g. `(row-cdr (list (list 1 2) (list 3 4)))` returns `((3 4))`
`col-cdr`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times (m-1)}$	Remove a matrix's first column, e.g. replacing `row-cdr` with `col-cdr` in the previous example returns `((2) (4))`
`row-cons`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{(n+1)\times m}$	Append a row to a matrix, e.g. `(row-cons (list 1 2) (list (list 3 4)))` returns `((1 2) (3 4))`
`col-cons`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times (m+1)}$	Append a column to a matrix, e.g. `(col-cons (list 1 3) (list (list 2) (list 4)))` returns `((1 2) (3 4))`
`m+`	$\mathbb{R}^{n\times m} \times \mathbb{R}^{n\times m} \to \mathbb{R}^{n\times m}$	Matrix addition, e.g. `(m+ (list (list 0 1) (list 2 3)) (list (list 2 3) (list 4 5))` returns `((2 4) (6 8))`
`m*`	$\mathbb{R}^{n\times m} \times \mathbb{R}^{m\times \ell} \to \mathbb{R}^{n\times \ell}$	Matrix multiplication, e.g., replacing `m+` with `m*` in the previous example returns `((4 5) (16 21))`
`sm*`	$\mathbb{R} \times \mathbb{R}^{n\times m} \to \mathbb{R}^{n\times m}$	Scalar-matrix multiplication, e.g. `(sm* 2 (list (list 1 2) (list 3 4)))` returns `((2 4) (6 8))`
`sv*`	$\mathbb{R} \times \mathbb{R}^{n} \to \mathbb{R}^{n}$	Scalar-vector multiplication, e.g. `(sv* 2 (list 1 2))` returns `(2 4)`
`out-*`	$\mathbb{R}^{n} \times \mathbb{R}^{n} \to \mathbb{R}^{n\times n}$	Outer product, e.g. `(out-* (list 1 2) (list 3 4))` returns `((3 4) (6 8))`
`get-L`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times m}$	Get a matrix's lower triangular part, e.g. `(get-L (list (list 1 2) (list 3 4)))` returns `((1 0) (3 4))`
`mtrans`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{m\times n}$	Matrix transpose, e.g. `(mtrans (list (list 1 2)))` returns `((1) (2))`

Program 1 ACL2 implementation of a Cholesky decomposition algorithm (Algorithm 1).

```
(define chol ((A matrixp))
 :guard (and (equal (col-count A) (row-count A))
             (equal (mtrans A) A))
 :measure (and (col-count A) (row-count A)) ...
 (mbe
 :logic
  (b* (;; BASE CASES
       ((unless (mbt (matrixp A))) (m-empty))    ;; If A not a matrix, return empty
       ((if (m-emptyp A)) A)                     ;; If A empty, return A
       (alph (car (col-car A)))                  ;; alph := "top left" scalar in A
       ((unless (realp alph))                    ;; If alph not real, return a zero
                (mzero (row-count A)             ;;   matrix of the same dimensions
                       (col-count A)))           ;;   as A
       ((if (<= alph 0))                         ;; If alph not positive, return a
        (mzero (row-count A)                     ;;   zero matrix of the same
               (col-count A)))                   ;;   dimensions as 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 ]
       (alph (acl2-sqrt alph))                   ;; alph := sqrt(alph)

       ;; BASE CASES
       ((if (m-emptyp (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-emptyp (row-cdr A)))              ;; If A is a row, return
        (row-cons (cons alph a12)                ;;   [ alph a12 ]
                  (m-empty)))

       ;; UPDATE
       (a21 (sv* (/ alph) a21))                  ;; a21 := a21 / alph
       (A22 (m+ A22 (sm* -1 (out-* a21 a21)))))  ;; A22 := A22 - a21 * a21T

      ;; RECURSE                                 ;;  [ alph | a12       ]
      (row-cons (cons alph a12)                  ;;  [ ---------------- ]
                (col-cons a21 (chol A22))))      ;;  [ a21  | CHOL(A22) ]
  :exec
   (b* ... ) ...) ...)
```

in the logic. In Program 1, the logic of `chol` is required to handle cases where it is passed non-matrix objects. Extra branches require more computational resources. To alleviate some of this overhead, we can use *guards* to prevent the *execution* of functions on unintended inputs. The macro `define` is a wrapper for `defun` that simplifies common hygienic practices, such as using guards, when introducing new ACL2 functions. In Program 1, the guards for `chol` are

```
(and (equal (col-count A) (row-count A))
     (equal (mtrans A) A))
```

as indicated by the `:guard` key and `(matrixp A)` as indicated by the formal arguments to `chol`. We deploy guard checking to prevent code execution under inappropriate or unexpected circumstances, helping catch potential issues early during program execution and avoiding unintended consequences, thus enhancing the robustness and reliability of ACL2 code. Another advantage of providing guards is it can reduce the computation performed by the Lisp back-end. For example, if `A` is known to be `matrixp`, then we no longer need

to perform the first check in the `b*` macro. If we know `A` is square, then we can reduce the number of base cases. The `mbe` macro enables a user to introduce a function logically defined by the term passed to the `:logic` key but executed with the code passed to the `:exec` key when the guards are satisfied. Of course, it must be proven that the logical definition and executed code are equivalent under these conditions.

4.2 Modifying Sylvester's Criterion

Recall that Theorems 1 and 2 hypothesize A to be symmetric positive definite. The usual definition for positive definiteness states that $v^T Av > 0$ for all nonzero $v \in \mathbb{R}^n$, which is a quantified statement. In symbols, this is

$$\forall v \in \mathbb{R}^n,\ v \neq 0 \implies v^T Av > 0\ .$$

This condition is not optimal for our theorem proving needs. On one hand, variables in ACL2 theorem statements are implicitly universally quantified at the top level. Quantifiers are also further supported via Skolem functions. However, Skolem functions are not executable. We want a recognizer for positive definite matrices to be executable because it can serve as guard for future functions,[4] and an executable recognizer more readily triggers the automatic rewrite rules which enable us to verify the Cholesky decomposition.

Instead of using the typical definition of positive definiteness, we use a definition which involves looking at the *leading principal submatrices*. Informally, the leading principal submatrices of a matrix A are the "top left" submatrices of size $k \times k$ for $k \in [1, n]$. For example, if a square matrix A is partitioned as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

and A_{11} is $k \times k$, then A_{11} would be the k-th leading principal submatrix of A. This approach is particularly useful for our computational purposes because it enables us to exploit the block structures of a matrix and restate matrix properties in terms of the same properties on smaller submatrices, such as determinants by Schur's formula.

▶ **Proposition 3** (Schur's formula). *Let* $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$. *Then*

$$\det(A) = \det(A_{11}) \det\left(A_{22} - A_{21}A_{11}^{-1}A_{12}\right)$$

if A_{11} *is invertible. Similarly,*

$$\det(A) = \det(A_{22}) \det\left(A_{11} - A_{12}A_{22}^{-1}A_{21}\right)$$

if A_{22} *is invertible. [4]*

Schur's formula enables us to use the following alternate definition for positive definiteness.

▶ **Definition 4** (Sylvester's criterion). *A symmetric matrix is positive definite iff all its leading principal submatrices have positive determinants.*

[4] We could also place ACL2's Cholesky decomposition implementation in a wrapper function with a recognizer for symmetric positive matrices as a guard.

Algorithm 2 An algorithm for checking whether symmetric matrices are positive definiteness.

procedure PD(A)

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** ▷ Base case

return True

if $\alpha \leq 0$ **then** ▷ Check if determinant is nonpositive

return False

return PD($A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^T$) ▷ Recursive case

This is also a quantified statement with the added issue that determinants are known to be computationally uncooperative. To avoid quantifiers, note the $(k-1)$-th leading principal submatrix of A is also the $(k-1)$-th leading principal submatrix of the k-th leading principal submatrix of A. The positivity of the former affects the positivity of the latter. This line of reasoning leads to a recursive (and, in particular, executable) recognizer.

Instead of explicitly computing determinants, we recursively check the positivity of the leading principal submatrix's determinant as shown in Algorithm 2. To understand this algorithm intuitively, consider the following partition

$$A = \begin{pmatrix} A_{11} & a_{12} & A_{13} \\ a_{21}^T & \alpha_{22} & a_{23}^T \\ A_{31} & a_{32} & A_{33} \end{pmatrix}.$$

Suppose A_{11} is the $(k-1)$-th leading principal submatrix of A and suppose we know $\det(A_{11}) > 0$. Then the determinant of the k-th leading principal submatrix of A is

$$\det \begin{pmatrix} A_{11} & a_{12} \\ a_{21}^T & \alpha_{22} \end{pmatrix} = \det(A_{11}) \det\left(\alpha_{11} - a_{21}^T A_{11}^{-1} a_{12}\right) = \det(A_{11}) \left(\alpha_{11} - a_{21}^T a_{12} / \det(A_{11})\right) \quad (2)$$

is positive iff $\alpha_{11} - a_{21}^T a_{12}/\det(A_{11}) > 0$. In Algorithm 2, this check is performed immediately after the recursive call.

To see an (informal) mathematical proof for why Algorithm 2 works, we require one more result.

▶ **Proposition 5.** *Suppose A is symmetric and partition $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$. Then A is positive definite iff A_{11} is positive definite and $A_{22} - A_{21}A_{11}^{-1}A_{12}$ is positive definite. [4]*

We are now ready to (informally) prove the correctness of Algorithm 2.

▶ **Theorem 6.** *Algorithm 2 returns "True" on a symmetric matrix A iff every leading principle submatrix of A has a positive determinant.*

Proof. To see the forwards direction, proceed by induction on the size n of A. For $n = 1$, we have $A = (\alpha_{11}) > 0$ when Algorithm 2 recognizes $\alpha > 0$. Let $n = k$ and suppose Algorithm 2 recognizes positive definiteness on symmetric matrices of size $k - 1$. Partition

$$k = \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix}.$$

Program 2 ACL2 function for checking positive definiteness.

```
(define positive-definite-p ((A matrixp))
 :guard (equal (col-count A) (row-count A))
 :measure (and (row-count A) (col-count A))
 :returns (pd booleanp)
 (b* (;; BASE CASES
      ((unless (matrixp A)) nil)                    ;; If A not a matrix, return empty
      ((if (m-emptyp A)) t)                         ;; If A empty, return A

      ;; CHECK IF DETERMINANT SO FAR IS POSITIVE
      (alph (car (col-car A)))                      ;; alph := "top left" scalar in A
      ((unless (realp alph)) nil)                   ;; If alph not real, return nil
      ((unless (< 0 alph))   nil)                   ;; If alph not positive, return nil

      ;; BASE CASES
      ((if (m-emptyp (row-cdr A))) t)               ;; If A is a row, return t
      ((if (m-emptyp (col-cdr A))) t)               ;; If A is a column, return t

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

      ;; COMPUTE THE SCHUR COMPLEMENT
      (alph    (acl2-sqrt alph))
      (a12     (sv* (/ alph) a12))
      (a21     (sv* (/ alph) a21))
      (A22     (m+ A22 (sm* -1 (out-* a12 a21))))) ;; A22 := A22 - a12 * a21T / alph

     ;; RECURSE
     (positive-definite-p A22))                     ;; Check if A22 is positive definite
 /// ...)
```

Thanks to Proposition 5, A is positive definite iff α_{11} and $A_{22} - a_{21}\alpha_{11}^{-1}a_{21}^T$ are both positive definite. If Algorithm 2 returns "True" on A, then we must have $\alpha_{11} > 0$ and $\mathrm{PD}\left(A_{22} - a_{21}\alpha_{11}^{-1}a_{21}^T\right)$ is true. Clearly, α_{11} is positive definite. Note A_{22} and $a_{21}\alpha_{11}^{-1}a_{21}^T$ are both symmetric and $(k-1) \times (k-1)$. By the induction hypothesis, $A_{22} - a_{21}\alpha_{11}^{-1}a_{21}^T$ is also positive definite.

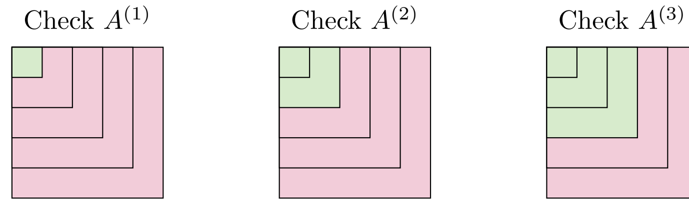
To see the other direction, we prove the contrapositive. Suppose Algorithm 2 returns "False" after k recursive calls, i.e. Algorithm 2 returned "True" $k-1$ times. Then some α_{11} must have been zero or negative. But we've already seen from Equation (2) that α_{11} is a factor in the determinant of the k-th leading principal submatrix. Since Algorithm 2 returned "True" all the other $k-1$ times, any other factor in the expansion via Schur's formula must be positive. Thus the determinant of the k-th leading principal submatrix must be zero or negative. ◀

The ACL2 implementation of Algorithm 2 is shown in Program 2. Visually, the progress of Program 2 is demonstrated in Figure 2. The structure of the program is similar to that of Program 1 in that it also progresses diagonally from the "top left" to the "bottom right". This similarity enables certain rewrite rules to fire and automatically verify the correctness of Program 1.

4.3 Verifying the Decomposition Algorithm

Given an executable and recursive condition for positive definiteness, we are now ready to formally prove Theorem 1 in ACL2. However, let's first look at the informal mathematical proof.

Figure 2 Progress of Program 2; $A^{(k)}$ denotes the k-th leading principal submatrix.



Program 3 ACL2 theorem for the correctness of a Cholesky decomposition program (Program 1).

```
(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))))
```

▶ **Theorem 1.** *Let A be a symmetric positive definite matrix. Let L be the lower triangular part of* CHOL(A)*. Then $A = LL^T$.*

Proof. Verifying the correctness of Program 1 amounts to induction on the size of A. The case where $n = 1$ is straightforward. Suppose CHOL computes the Cholesky decomposition for symmetric positive definite matrices of dimension $(k-1) \times (k-1)$. To see that CHOL computes the appropriate decomposition when A is $k \times k$, we just need to unravel one "layer" of LL^T and apply our induction hypothesis. Again partition

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

so that

$$\text{CHOL}(A) = \begin{pmatrix} \sqrt{\alpha_{11}} & a_{12}^T \\ a_{21}/\sqrt{\alpha_{11}} & \text{CHOL}\left(A_{22} - a_{21}a_{12}^T/\alpha_{11}\right) \end{pmatrix}$$

according to Algorithm 1. Let L be the lower triangular part of CHOL(A) and let L_{22} be the lower triangular part of CHOL$(A_{22} - a_{21}a_{12}^T/\alpha_{11})$. Since $A_{22} - a_{21}a_{12}^T/\alpha_{11}$ is a symmetric positive definite $(k-1) \times (k-1)$ matrix, the lower triangular part of CHOL on the same matrix is its own Cholesky decomposition, i.e. $L_{22}L_{22}^T = A_{22} - a_{21}a_{12}^T/\alpha_{11}$. Then

$$\begin{aligned} LL^T &= \begin{pmatrix} \sqrt{\alpha_{11}} & \\ a_{21}/\sqrt{\alpha_{11}} & L_{22} \end{pmatrix} \begin{pmatrix} \sqrt{\alpha_{11}} & \\ a_{21}/\sqrt{\alpha_{11}} & L_{22} \end{pmatrix}^T \\ &= \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & L_{22}L_{22}^T + a_{21}a_{21}^T/\alpha_{11} \end{pmatrix} \\ &= \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix} \\ &= A \end{aligned} \tag{3}$$

as desired. ◀

Program 3 contains the ACL2 theorem for verifying the correctness of Program 1. In addition to `(positive-definite-p A)`, the hypothesis posits that $A = A^T$ and that A is square. The extra symmetry condition is necessary because `positive-definite-p` in

Program 4 ACL2 Skolem function positing the existence of an LU decomposition.

```
(defun-sk chol-fact-exists (A)
  (exists (L) (and (lower-tri-p L) (equal (m* L (mtrans L)) A))))
```

Program 5 Theorem event automatically introduced into ACL2 by `defun-sk` in Program 4.

```
(defthm chol-fact-exists-suff
  (implies (and (lower-tri-p L)                ;; L is lower triangular
                (equal (m* L (mtrans L)) A))    ;; L * L^T = A
           (chol-fact-exists A)))               ;; A has a Cholesky decomposition
```

Program 2 doesn't assume that `a12` is equal to `a21`. Similarly, Equation (3) in the proof of Theorem 1 requires $a_{21} = a_{12}$ in order to recover A. Ultimately, the similarity between the structure of `positive-definite-p` in Program 2 and `chol` in Program 1 is what enables us to discharge `chol-correctness` in Program 3 automatically.

4.4 From Decomposition Algorithm to Factorization Theorem

Theorem 1 is distinct from statement of Theorem 2, which posits the existence of a Cholesky decomposition in its conclusion.

▶ **Theorem 2** (Cholesky Factorization Theorem). *If A is a symmetric positive definite matrix, then $A = LL^T$ for some lower triangular matrix L.*

Reasoning in ACL2 typically takes place by making propositional statements about functions, such as Program 3 or, equivalently, Theorem 1. One advantage to this is that ACL2 is highly automated. One disadvantage to this is that expressing statements such as Theorem 2 can be a challenge.

While the ACL2 logic is quantifier-free, reasoning about quantified statements is still supported by way of *Skolem functions*. A Skolem function in ACL2, introduced by `defun-sk`, is a function whose body has an outermost quantifier. For example, an ACL2 Skolem function for the latter part of Theorem 2 is Program 4. The function `chol-fact-exists` states "there exists an L such that L is lower triangular and $LL^T = A$". The function `lower-tri-p` is simply a recognizer for lower triangular matrices, the ACL2 code for which we omit for brevity. To state Theorem 2 then amounts to placing a call to `chol-fact-exists` in the conclusion of a typical ACL2 theorem. Strictly speaking, we are still reasoning by asserting a function within a propositional statement; the function simply describes a quantified statement. The specifics of `defun-sk` are beyond the scope of this paper. The upshot is that a theorem of the form seen in Program 5 is automatically introduced into the ACL2 logical universe. Essentially, the theorem `chol-fact-exists-suff` states that if `L` is lower triangular and `L` multiplied by its transpose equals `A`, then there exists a Cholesky decomposition for `A`. Ultimately, we want to eliminate the hypotheses involving `L` and have the conclusion hold conditioned purely on `A`. If a witness is provided for `L`, then we can prove Theorem 2. Given Program 3, the clear witness is the lower triangular part of `(chol A)`. We pass the witness by instantiating `chol-fact-exists-suff` and replacing `L` with `(get-L (chol A))` using a hint in the desired theorem. The desired theorem is Program 6.

Program 6 The Cholesky Factorization Theorem in ACL2.

```
(defthm cholesky-factorization-theorem
 (implies (and (equal (mtrans A) A)
               (positive-definite-p A)
               (equal (col-count A) (row-count A)))
          (chol-fact-exists A))
 :hints (("Goal" :use ((:instance chol-fact-exists-suff (L (get-L (chol A))))))))
```

Table 3 ACL2 statistics related to the Cholesky Factorization Theorem.

Lines of code	1140
Events	192
Prover steps	5 029 675
Verification time (s)	7.91
Memory allocated (GB)	1.37

5 Conclusion

In this paper, we formalize and verify a Cholesky decomposition algorithm. Our work is open sourced as an ACL2 “book”, which can be found in the ACL2 GitHub repository [12] and as part of future ACL2 releases [11]. We invite interested readers to try decomposing their own matrices using ACL2. A summary of ACL2 statistics for this work is shown in Table 3. Events are updates to the ACL2 logical world, such as new definitions or theorems. Prover steps are the number of steps to justify the events. Verification was performed on a laptop with an Apple M1 Pro CPU.

Few theorem prover formalizations of numerical linear algebra algorithms exist; this is likely because typical numerical algorithms heavily employ indexing and few theorem provers are equipped to reason in this manner. We embed the FLAME environment into ACL2 so that we may verify the veracity of such algorithms. FLAME separates itself from other approaches by presenting algorithms which are designed to be proven correct, in no small part due to how matrices are partitioned in the derivation. The FLAME approach facilitates a useful derivation that enables us to develop an elegant ACL2 proof for the Cholesky Factorization Theorem.

Our Cholesky decomposition algorithm takes square roots in each recursive update which is why we use ACL2(r). Otherwise, using the vanilla version of ACL2 (without support for real and complex irrationals) would be sufficient. The presentation in this paper used our logical definition of Cholesky, which involves taking square roots by way of operations involving a nonstandard objects. To make execution more amenable, we define an alternate Cholesky program which employs an iterative square root function `sqrt-iter`. This iterative square root has been verified to converge to the logical square root `acl2-sqrt` [7]. It is also possible to reason about square roots in vanilla ACL2 using only its algebraic properties, e.g. by augmenting the field of ACL2 numbers with $\sqrt{\ }$. Instead of developing a new theory in ACL2, we decided to simply use ACL2(r). Moreover, a theory of infinitesimals, such as the one supported by the non-standard analysis in ACL2(r), would be useful for any future attempt at verifying the *backwards error analysis* of formalized numerical linear algebra algorithms.

One major future direction for our work is to develop an ACL2 framework for reasoning about backwards error analysis. Backwards error analysis is paramount to ensuring the stability of numerical algorithms, thus providing a form of safety to their critical applications.

Recent ACL2 developments have enabled support for ACL2 computations involving floating-point numbers [1], providing us with an appropriate framework to reason about floating-point implementations of numerical algorithms. Moreover, expressing stability involves taking the norms of vectors and matrices, which motivates our future ACL2 investigation into these topics.

In addition to formalizing stability, we want to develop an ACL2 theory of norms because they are used in other numerical algorithms which deserve verification, such as QR decomposition. The QR decomposition is another fundamental algorithm in scientific computing with numerous applications and is usually introduced by the Gram-Schmidt process. However, we anticipate the *Householder* QR decomposition algorithm, a more stable alternative to Gram-Schmidt, to share the structure of Algorithm 1. The similar structures suggest that Householder QR may find an ACL2 verification in much the same way as our Cholesky decomposition algorithm in this paper. We will target a Householder QR decomposition algorithm for verification as future work.

Thus far we have discussed the use of ACL2 as a proof assistant for verifying theorems and formalizing theories. Indeed, the relatively low ratio of lines of code to theorem prover steps from Table 3 indicates that ACL2 has a high degree of automation in the context of proving pure mathematical results. But another future direction is to use our work (for not just the verification of, but also) *in* scientific computing's most critical applications, which span fields such as machine learning, medical imaging, bioengineering, finance, structural engineering, aerospace, and much more. To make a verified numerical linear algebra library practical, it also needs to be efficient. A concrete optimization we can make in Algorithm 1 is to ignore the strictly upper triangular part of A. Note that because A is assumed to be symmetric and only symmetric updates are performed, namely $A_{22} \coloneqq A_{22} - a_{21}a_{21}^T$, there is no need to update both the lower triangular and the upper triangular parts. Moreover, once the algorithm terminates, we are only interested in the lower triangular part of the result. Instead, we can update merely the lower triangular part of $A_{22} \coloneqq A_{22} - a_{21}a_{21}^T$, performing a so-called *symmetric rank-one update*, to halve the computational cost of the algorithm.

Improvements can also be made to improve the baseline execution speed of our matrix programs. By default, numeric computations are offloaded to Lisp. Significant efforts, now part of the standard ACL2 toolbox, have been made to improve the executional efficiency of certain kinds of models. One such improvement is the development of *single-threaded objects* (stobjs) [5]. Logically, a stobj is a standard ACL2 association list; on the backend, updates to a stobj are made via destructive memory assignments, making them highly practical in situations where execution speed is vital. Stobjs were originally developed to improve the simulation speeds of ACL2 microprocessor models. Given our simple "list of lists" model of matrices, stobjs and its supporting framework, such as "access" and "update" functions, can also be used to represent ACL2 matrices and matrix operational semantics, respectively. The efficiency of destructive assignments in memory during field updates is ample motivation to investigate the potential use of stobjs for numerical linear algebra algorithms, especially those which involve computing a result in place, such as the Cholesky decomposition algorithm we formalize in this paper.

There are very few theorem prover formalizations of numerical linear algebra algorithms – even fewer have support for execution. Developing non-executable libraries of numerical algorithms (formal or otherwise) seems antithetical to their computational nature. In addition to presenting the first formalization of a major theorem in linear algebra, our work in this paper sets a precedent for verified scientific computing in the future.

References

1 ACL2. *Df: Support for floating-point operations.* Accessed 2024-06-06. URL: https://cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/ACL2____DF.

2 ACL2. *Loop$.* Accessed 2024-06-06. URL: https://cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html?topic=ACL2____LOOP_42.

3 ACL2. *User manual for the ACL2 Theorem Prover and the ACL2 Community Books.* Accessed 2023-07-12. URL: https://cs.utexas.edu/users/moore/acl2/current/manual/index.html.

4 Stephen Boyd and Lieven Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004. doi:10.1017/CBO9780511804441.

5 Robert S. Boyer and J. Strother Moore. Single-threaded objects in ACL2. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 9–27, Berlin, Heidelberg, 2002. Springer-Verlag. doi:10.5555/645772.667953.

6 Ruben Gamboa, John Cowles, and Jeff Van Baalen. Using ACL2 arrays to formalize matrix algebra, 2003. URL: https://cs.uwyo.edu/~ruben/static/pdf/matalg.pdf.

7 Ruben A. Gamboa and Matt Kaufmann. Nonstandard analysis in ACL2. *J. Autom. Reason.*, 27(4):323–351, November 2001. doi:10.1023/A:1011908113514.

8 John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, December 2001. doi:10.1145/504210.504213.

9 John Harrison. The HOL Light theory of Euclidean space. *Journal of Automated Reasoning*, 50:173–190, 2012. doi:10.1007/s10817-012-9250-9.

10 Joe Hendrix. Matrices in ACL2, 2003. URL: https://cs.utexas.edu/users/moore/acl2/workshop-2003/contrib/hendrix/hendrix.pdf.

11 Matt Kaufmann and J Strother Moore. ACL2 home page. https://cs.utexas.edu/~moore/acl2/, 1997. Accessed 2024-06-25.

12 Matt Kaufmann and J Strother Moore. ACL2 system and community books. https://github.com/acl2/acl2, 2014. Accessed 2024-06-25.

13 Carl Kwan. Classical LU decomposition in ACL2. *Electronic Proceedings in Theoretical Computer Science*, 393:1–5, November 2023. doi:10.4204/eptcs.393.1.

14 Carl Kwan and Mark R. Greenstreet. Real vector spaces and the Cauchy-Schwarz inequality in ACL2(r). *Electronic Proceedings in Theoretical Computer Science*, 280:111–127, October 2018. doi:10.4204/eptcs.280.9.

15 Lean mathlib3 documentation: LDL decomposition. https://leanprover-community.github.io/mathlib_docs/linear_algebra/matrix/ldl.html. Accessed 2023-07-13.

16 Lean mathlib3 documentation: Matrices. https://leanprover-community.github.io/mathlib_docs/data/matrix/basic.html. Accessed 2023-07-13.

17 ZhengPu Shi and Gang Chen. Integration of multiple formal matrix models in Coq. In Wei Dong and Jean-Pierre Talpin, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 169–186, Cham, 2022. Springer Nature Switzerland. doi:10.1007/978-3-031-21213-0_11.

18 Zhiping Shi, Yan Zhang, Zhenke Liu, Xinan Kang, Yong Guan, Jie Zhang, and Xiaoyu Song. Formalization of matrix theory in HOL4. *Advances in Mechanical Engineering*, 6:195–276, 2014. doi:10.1155/2014/195276.

19 Christian Sternagel and René Thiemann. Executable matrix operations on matrices of arbitrary dimensions. *Archive of Formal Proofs*, June 2010. , Formal proof development. URL: https://isa-afp.org/entries/Matrix.html.

20 René Thiemann and Akihisa Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, August 2015. , Formal proof development. URL: https://isa-afp.org/entries/Jordan_Normal_Form.html.