LAProof: A Library of Formal Proofs of Accuracy and Correctness for Linear Algebra Programs

Ariel E. Kellison *, §, Andrew W. Appel †, Mohit Tekriwal ‡ and David Bindel *

* Dept. of Computer Science, Cornell University, Ithaca, NY, USA
† Dept. of Computer Science, Princeton University, Princeton, NJ, USA
‡ Dept. of Aerospace Engineering, University of Michigan, Ann Arbor, MI, USA
§ Sandia National Laboratories, Livermore, California, USA

Abstract—The LAProof library provides formal machine-checked proofs of the accuracy of basic linear algebra operations: inner product using conventional multiply and add, inner product using fused multiply-add, scaled matrix-vector and matrix-matrix multiplication, and scaled vector and matrix addition. These proofs can connect to concrete implementations of low-level basic linear algebra subprograms; as a proof of concept we present a machine-checked correctness proof of a C function implementing sparse matrix-vector multiplication using the compressed sparse row format. Our accuracy proofs are backward error bounds and mixed backward-forward error bounds that account for underflow, proved subject to no assumptions except a low-level formal model of IEEE-754 arithmetic. We treat low-order error terms concretely, not approximating as \( O(u^2) \).

Index Terms—rounding error analysis, formal verification, floating-point arithmetic, program verification

I. INTRODUCTION

Numerical linear algebra is widely used across computational disciplines and is serving an increasingly important role in emerging applications for embedded systems. The Basic Linear Algebra Subprograms (BLAS) [1], [2] provide a modular, reliable standard defining a set of the most common linear algebra operations such as the inner product and the matrix-vector product. The software layer implementing the operations defined by BLAS is often highly optimized and architecture-specific, serving as an interface between hardware and application software. While implementations of BLAS may differ in practice, an implementation should guarantee numerical accuracy with respect to widely accepted rounding error bounds. In this paper, we report on our development of the Linear Algebra Proof Library (LAProof), a library of formal proofs of rounding error analyses for basic linear algebra operations. LAProof serves as a modular, portable proof layer between the verification of application software and the verification of programs implementing operations defined by BLAS. The LAProof library makes the following contributions:

- **Backward and mixed backward-forward error bounds.** Previous formal rounding error analyses have exclusively focused on forward error bounds (see related work in Section V). We provide backward and mixed backward-forward error bounds. This choice is advantageous from the perspectives of both proof engineering and numerical analysis, as it preserves the separation of rounding errors from the structural conditions of the mathematical problem being solved by the application software. Furthermore, forward error bounds can be derived directly from backward and mixed backward-forward error bounds.
- **No linearization of error terms.** The rounding error associated with a sequence of operations accumulates errors as products of terms of the form \( (1 + \delta_i) \), where the magnitude of each \( \delta_i \) is uniformly upper-bounded by the unit roundoff, \( u \). Typically, numerical analysts simply linearize the product of these terms, approximating the error in \( n \) operations by \( nu + O(u^2) \). In LAProof we avoid such approximations, giving clients of the library access to error analyses that fully characterize the accumulation of error in any sequence of operations.
- **Minimal assumptions and soundness.** The LAProof library is fully developed inside of the Coq proof assistant, and assumes only the Flocoq [3] specification of the IEEE 754 standard [4] for floating-point arithmetic. LAProof’s rounding error analysis is therefore sound with respect to the IEEE standard. Furthermore, the error bounds provided by LAProof do not assume the absence of underflow; and where the proofs assume the absence of overflow, we provide a concrete example of how this assumption can be discharged for operations where numerical bounds on the terms in a linear algebra expression are known (see Section IV).
- **Connection to sparse matrix implementation in C.** To demonstrate that our accuracy theorems can be seamlessly composed with correctness proofs of programs that use nontrivial data structures, we use LAProof in the verification of a C program implementing sparse matrix-vector multiply using the compressed sparse row (CSR) format.

The remaining sections of the paper clarify the contributions of the LAProof library. Section II introduces the basic linear algebra operations provided by LAProof and describes their formal error bounds. Section III explains the implementations of the core LAProof operations in the Coq proof assistant, emphasizing their soundness with respect to the IEEE standard. Section IV demonstrates how the LAProof library can be used to guarantee the accuracy of concrete C programs using a machine-checked correctness proof of a C function.
implementing CSR matrix-vector multiplication. Section V situates the LAProof library with respect to related work, and Section VI discusses the current limitations of LAProof and future work.

II. OVERVIEW OF THE LIBRARY

The LAProof library provides formal proofs of error bounds derived from well understood error analyses [5], [6] for the basic linear algebra operations listed in Tables I, II, and III. The error bounds for each operation are parameterized by the precision of the standard IEEE 754 floating-point formats supported by the Flocq library [3], and are derived using the standard rounding error model for floating-point arithmetic [6, sect 2.2]:

$$\fl(a \ op b) = (a \ op b)(1 + \delta) + \epsilon$$

(1)

$$|\delta| \leq u, \ |\epsilon| \leq \eta, \ \delta \epsilon = 0, \ op \in \{+, -, \times, /, \sqrt{\}$$

Our formal error bounds are derived in Coq, relying on the Flocq library’s machine-checked proofs [3] that this standard error model holds for floating point arithmetic. Throughout the paper, for a floating-point number with precision p and maximum exponent e, we denote the unit roundoff by u = 2−p, the underflow threshold by \(\eta = 2^{2^{e+p}}\), and the exactly representable numbers in the format by \(F_{p,e}\).

The LAProof error analysis for each operation is performed by writing two pure functional programs in Gallina, the functional programming language embedded in Coq: a real-valued function \(\phi\) represents the operation in exact arithmetic, and a floating-point valued function \(\phi_{p,e}\) defined over the IEEE 754 format specified by Flocq. Using these functional programs, the absolute forward error F is expressed as

$$F \triangleq |\phi_{p,e}(x) - \phi_{p,e}(x)|.$$  

(2)

The mixed backward-forward error requires deriving a suitable perturbation \(\Delta x\) to the inputs of \(\phi_{p,e}\) and small forward error term \(\dot{\delta}\) such that

$$\phi_{p,e}(x) = \phi_{p,e}(x + \Delta x) + \dot{\delta}.$$  

(3)

The error bounds in LAProof are expressed using the functions \(h(n)\) and \(g(n, m)\) to represent the accumulation of error from rounding normal and denormal numbers, respectively:

$$h(n) = (1 + u)^n - 1.$$  

(4)

$$g(n, m) = n\eta(1 + h(m)).$$  

(5)

A. Vector Operations

The core vector operations in LAProof are the inner (dot) product \(r \leftarrow x \cdot y\), vector addition \(r \leftarrow x + y\), summation \(r \leftarrow \sum_i x_i\), and scaling by a constant \(r \leftarrow \alpha x\). We provide mixed backward-forward error bounds for the inner product and scaling by a constant. Given that addition and subtraction are exact for denormal numbers [7], we provide a strict backward error bound for summation and vector addition. Error analyses for scaled vector addition \(r \leftarrow \alpha x + \beta y\) and the vector norms listed in Table I follow by composing error bounds of the core vector operations. In the remainder of this section, we sketch the error analyses formalized in LAProof for the inner product and summation, and discuss some useful corollaries.

TABLE I

<table>
<thead>
<tr>
<th>LAPROOF VECTOR OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOT</td>
</tr>
<tr>
<td>sVec</td>
</tr>
<tr>
<td>SUM</td>
</tr>
<tr>
<td>VecAdd</td>
</tr>
<tr>
<td>VecAXPY</td>
</tr>
<tr>
<td>VecNRM1</td>
</tr>
<tr>
<td>VecNRM2</td>
</tr>
</tbody>
</table>

LAProof provides a formal proof of the following mixed backward-forward error bound for the inner product of two vectors assuming the absence of overflow.

**Theorem 1 (bfDOT).** For any two vectors \(u, v \in F_{p,e}^n\), the vectors \(\hat{u}, \delta \in \mathbb{R}^n\) and real number \(c \in \mathbb{R}\) exist such that

$$\fl(\hat{u} \cdot \hat{v}) = \hat{u} \cdot v + c,$$

where \(|c| \leq g(n, n)\) and every \(k^{th}\) element of \(\hat{u}\) respects the bound \(\hat{u}_k = u_k(1 + \delta_k)\) with \(|\delta_k| \leq h(n)\).

**Proof.** The most common exact-arithmetic version of the inner product computation loops over the elements of \(u\) and \(v\) to accumulate the partial sums \(s_k = s_{k-1} + u_k v_k\), starting from \(s_1 = u_1 v_1\). In floating point, we have \(\hat{s}_1 = u_1 v_1(1 + \delta_1) + \epsilon_1\) for \(k = 1\), and

$$\hat{s}_k = (\hat{s}_{k-1} + u_k v_k (1 + \delta_k) + \epsilon_k)(1 + \gamma_k)$$

(7)

with \(|\delta_k| \leq u, |\gamma_k| \leq u, \text{ and } |\epsilon_k| \leq \eta\). If we define \(\gamma_1 = 0\), we have

$$\hat{s}_k = \sum_{j=1}^k \left[ u_j v_j (1 + \delta_j) + \epsilon_j \prod_{\ell=j}^k (1 + \gamma_\ell) \right].$$

Now define

$$\hat{\gamma}_j = \sum_{j=1}^n (1 + \gamma_\ell) - 1, \quad \hat{\delta}_j = (1 + \delta_j)(1 + \hat{\gamma}_j) - 1$$

for which we have the bounds

$$|\hat{\gamma}_j| \leq h(n - j) \leq h(n - 1), \quad |\hat{\delta}_j| \leq h(n - j + 1) \leq h(n).$$

The computed dot product is

$$\hat{s}_n = \sum_{j=1}^n u_j v_j (1 + \hat{\delta}_j) + \sum_{j=1}^n \epsilon_j (1 + \hat{\gamma}_j),$$

or, equivalently

$$\hat{s}_n = \hat{u} \cdot \hat{v} + c.$$
where \(|c| \leq g(n, n)\) and \(\hat{u}_j = u_j(1 + \delta_j)\). This is a mixed error bound because it combines a backward error term \(\hat{u}_j\) and a forward error term \(c\).

In the absence of underflow, using a linear approximation to the error function \(h\) reduces the bound in Theorem 1 to that given in the literature [6, sec 3.1].

We prove the following as corollaries to Theorem 1.

a) Forward error: Given equation (6), the forward error bound

\[
\|f((\mathbf{u} \cdot \mathbf{v}) - (\mathbf{u} \cdot \mathbf{v}))\| \leq (S(\mathbf{u}, \mathbf{v})) h(n) + h(n, n - 1) \tag{8}
\]

where

\[
S(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} |x_i \cdot y_i| \tag{9}
\]

is straightforward to derive. Linearizing the error function \(h\) reduces the forward error bound to that given in the literature [8].

b) Sparsity: Assuming that one of the vectors \(\mathbf{u}, \mathbf{v} \in \mathbb{F}_{p,e}^n\) is sparse, we prove the theorem \(\text{sbDOT}\), in which the error functions \(g\) and \(h\) in theorem \(\text{bfDOT}\) are parameterized by the smallest number of nonzero elements occurring in either vector.

c) FMA: If the fused multiply-add operation is used to compute an inner product of the vectors \(\mathbf{u}, \mathbf{v} \in \mathbb{F}_{p,e}^n\), the floating-point partial sums in equation 7 become

\[
\hat{s}_k = \text{FMA}(u_k, v_k, \hat{s}_{k-1}) = (u_k v_k + \hat{s}_{k-1})(1 + \delta_k) + \epsilon_k
\]

where \(|\delta_k| \leq u\) and \(|\epsilon_k| \leq \eta\). Following a similar analysis to that of Theorem 1, we prove that the FMA inner product respects the error bounds given in equations 6 and 8.

For summing vector elements, the LAProof library provides the following backward error bound.

Theorem 2 (bSUM). For any vector \(\mathbf{u} \in \mathbb{F}_{p,e}^n\), the vector \(\hat{u} \in \mathbb{R}^n\) exists such that

\[
f(\sum_{i=1}^{n} u_i) = \sum_{i=1}^{n} \hat{u}_i, \tag{10}
\]

where every \(k^{th}\) element of \(\hat{u}\) respects the bound \(\hat{u}_k = u_k(1 + \delta_k)\) with \(|\delta_k| \leq h(n, n - 1)\).

d) Order of summation: Theorem 2 holds as an upper bound on the absolute forward error for any permutation of the vector \(\mathbf{u}\). At a low level, the formalization of vectors in LAProof uses Coq lists, for which is easy to prove this fact.

B. Matrix-Vector Operations

The core matrix-vector operation provided to clients of the LAProof library is the matrix-vector product \((r \leftarrow Ax\)). An error bound for the scaled matrix-vector product \((r \leftarrow \alpha Ax)\) is also provided. The error bound for matrix-vector multiplication follows from the mixed backward-forward error bound for the inner product given in the previous section, following the standard analysis [6, sec 3.5]. Assuming the absence of overflow, LAProof provides the following error bound for matrix-vector multiplication.

Theorem 3 (bfMV). For any vector \(\mathbf{u} \in \mathbb{F}_{p,e}^n\) and matrix \(\mathbf{M} \in \mathbb{F}_{p,e}^{m \times n}\), there exist a matrix \(\Delta \mathbf{M} \in \mathbb{F}_{p,e}^{m \times n}\) and vector \(\eta \in \mathbb{R}^n\) such that

\[
\|f(Mu) - (M + \Delta M)u + \eta\| \leq (\|M + \Delta M\|u + \eta)
\]

where every element of the backward error term \(\eta\) respects the bound \(|\eta| \leq g(n, n)\) and each element of the forward error term \(\Delta M\) respects the bound \(|\Delta M| \leq h(n)[M].

<table>
<thead>
<tr>
<th>TABLE II</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAProof Matrix-Vector Operations</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>MV</td>
</tr>
<tr>
<td>(r \leftarrow Ax)</td>
</tr>
<tr>
<td>sMV</td>
</tr>
<tr>
<td>(r \leftarrow \alpha Ax)</td>
</tr>
<tr>
<td>GEMV</td>
</tr>
<tr>
<td>(r \leftarrow \alpha Ax + \beta y)</td>
</tr>
</tbody>
</table>

a) Conditions for the absence of overflow: In Section IV we demonstrate how the LAProof implementation of matrix-vector multiplication can connect to concrete implementations of low-level basic linear algebra subprograms. In order to guarantee that the concrete implementation respects the error bound given in Theorem 3, the assumption of the absence of overflow must be discharged. The LAProof library guarantees the absence of overflow for matrix-vector multiplication under the following conditions.

Theorem 4 (finiteMV). For any vector \(\mathbf{u} \in \mathbb{F}_{p,e}^n\) and matrix \(\mathbf{M} \in \mathbb{F}_{p,e}^{m \times n}\), if the magnitude of each of the elements of \(\mathbf{u}\) and \(\mathbf{M}\) is upper bounded by the square root of

\[
\left(\frac{2e - \eta}{1 + u} - g(n, n - 1)\right) \left(\frac{1}{1 + n(1 + u)^n}\right)
\]

and \(g(n + 1, n) \leq 2e\), then the floating-point result of \(f(Mu)\) is a vector with elements in the floating-point format \(\mathbb{F}_{p,e}\).

The upper bound in Theorem 4 is achieved in many settings. As an example, for binary32 and \(n = 10e6\) the magnitude of the elements of \(\mathbf{u}\) and \(\mathbf{M}\) must be less than \(1.8e16\).

C. Matrix Operations

The core matrix operations in LAProof are the matrix-matrix product \((R \leftarrow AB)\), matrix addition \((R \leftarrow A + B)\), and scaling by a constant \((R \leftarrow \alpha A)\). The formal rounding error analysis for these operations follows from the mixed backward-forward error bounds for the matrix-vector product, vector addition, and vector scaling. The LAProof library provides a forward error bound for the matrix-matrix product following the literature [6], [9], a backward error bound for matrix addition, and a mixed backward-forward error bound for matrix scaling. Formal mixed error bounds are also provided for scaled matrix-matrix multiplication \((R \leftarrow \alpha AB)\), the addition of a scaled matrices \((R \leftarrow \alpha A + \beta Y)\), and a scaled matrix-matrix product plus a scaled matrix \((R \leftarrow \alpha AX + \beta Y)\).
III. Functional Models

LAProof operations are pure functional programs written in Gallina, the functional programming language embedded in Coq; we refer to these programs as functional models. These operations are Coq functions over Coq lists (or lists of lists) of either Coq’s axiomatic real numbers or a type ftype, which denotes IEEE 754 compliant binary floating-point formats with the precision and maximum exponent supplied by the argument t. The ftype function is provided by the VCFloat package [10], [11], and is a user-friendly wrapper around the Flocoq formalization of IEEE binary floating-point formats. We begin our description of the LAProof functional models by first introducing the low-level Coq definitions of matrices and vectors upon which they depend.

A. Matrices and vectors

Matrices and vectors in LAProof are defined using Coq lists over an arbitrary element type T.

Definition matrix (T : Type) := list (list T).
Definition vector (T : Type) := list T.

For example, a vector of double-precision floats in LAProof would have type vector (ftype TDouble). Henceforth, we will use the notation of \( \mathbb{R}^n \) and vector (ftype t) interchangeably, and similarly for matrices. The LAProof library is developed under the assumption that matrices are in row-major form.

B. Vector operations

The fundamental operation in the LAProof library is the inner product. In order to define real-valued and floating-point valued functional models for the inner product, it suffices to define a generic polymorphic function DOT over an arbitrary implicit type T. DOT takes two vectors u and v with elements of type T, and the functions add and mul, and produces a result of type T.

Variables add mul : T → T → T.
Variables u v : vector T.

Definition DOT : T :=
  fold_left add (map (uncurry mul) (combine u v)).

We define a floating-point functional model DOTF by supplying DOT with a generic ftype t type and the appropriate functions over this type; LAProof uses the VCFloat functions BPLUS and BMULT over ftype t. These VCFloat functions are simply wrappers around the corresponding IEEE operators defined in Flocoq, so LAProof inherits the soundness of these operators with respect to the IEEE specification formalized by Flocoq. A real-valued functional model DOTR is similarly defined with the addition and multiplication operations supplied by Coq’s theory of axiomatic reals. Using these functional models, the mixed backward-forward error bound given in Theorem 1 is stated in Coq as follows.

Variable t : type.
Variables u v: vector (ftype t).
Hypothesis Hfin: is_finite (DOTF u v) = true.
Let n := (length v).

Theorem bfDOT: \( \exists (u' : list \mathbb{R}) (\eta : \mathbb{R}), \)
FT2R (DOTF u v) = DOTR u’ (map FT2R v) + \eta
\( \land (\forall i : \mathbb{N}, (i < n) \rightarrow \exists h : \mathbb{R}. u'_i = (1 + \delta) \cdot FT2R(v_i) \)
\( \land |\delta| \leq h(n) \land |\eta| \leq g(n,n)) \).

where the function FT2R : ftype t → \mathbb{R} is an injection from the floating-point values of type ftype t to real values.

Floating-point and real-valued functional models for the remaining vector operations of addition, summation, and scaling by a constant are defined from the following polymorphic functions over a generic element type T.

Variables add mul : T → T → T.
Definition VecAdd (u v : vector T) : vector T :=
  map (uncurry add) (combine u v).
Definition SUM : vector T → T :=
  fold_right add.
Definition sVec (a: T) : vector T :=
  map (mul a).

C. Matrix-vector operations

The core matrix vector operation implemented in LAProof is the matrix-vector product. We denote the floating-point and real valued functional models for matrix-vector multiplication implemented by LAProof as MVF and MVR. These functions are built by supplying the previously defined inner products (DOTR and DOTF) to a polymorphic function MV defined over an arbitrary implicit element type T:

Variables dot : vector T → vector T → T.

Definition MV : vector T :=
  map (fun a => dot a v) A.

A formal statement of the mixed backward-forward error bound for matrix-vector multiplication given in Theorem 3 requires defining suitable functional models for matrix addition.

D. Matrix operations

Functional models for floating-point and real valued matrix addition and scaling by a constant are defined in LAProof from the following polymorphic functions.

| TABLE III |
| LAPROOF MATRIX OPERATIONS |
|----|----|
| sMat | \( R \leftarrow \alpha A \) |
| MM | \( R \leftarrow AB \) |
| MatAdd | \( R \leftarrow A + B \) |
| sMM | \( R \leftarrow \alpha AB \) |
| MatAXPB | \( R \leftarrow \alpha Ax + \beta Y \) |
| GEMM | \( R \leftarrow \alpha Ax + \beta Y \) |
Definition map2 {A B C: Type}(f:A→B→C)(x:A)(y:B) := map (uncurry f) (combine x y).

Definition sMat {T: Type} (mul: T→T→T) (a: T) := map (map mul a).

Definition MatAdd {T: Type} (sum: T→T→T) := map2 (map2 sum).

Denoting real-valued MatAdd and real-valued VecAdd as \(+_m\) and \(+_e\), respectively, and floating-point valued and real-valued MV as \(\odot\) and \(*_e\), respectively, the formal LAProof statement of Theorem 3 is given as follows.

Variable (A : matrix (ftype t)).
Variable (v : vector (ftype t)).

Let \(m := (\text{length} A)\).
Let \(n := (\text{length} v)\).

Notation \(\text{Ar} := (\text{map} (\text{map} \text{FT2R}) A)\).
Notation \(\text{vr} := (\text{map FT2R} v)\).

Hypothesis Hfin : \(\forall\) (E:matrix R) (\(\text{is_finite_vec}\) (MVF A v)).
Hypothesis Hlen : \(\forall\) (v: vector T), In \(\text{length} v\) \(\rightarrow\) length \(x\) = \(n\).

Theorem bMV: \(\exists\) (E:matrix R) (\(\forall\) (v : vector R)).
\(\text{map} \text{FT2R} (A \odot_v v) = (\text{Ar} \odot_m \text{vr} +_e \eta)\) \(\land\) \(\forall (i < m, j < n) \rightarrow |E_{i,j}| \leq h(n) |\text{Ar}_{i,j}|\) \(\land\) \(\forall (k : \mathbb{N}) \in k \eta \rightarrow |k| \leq g(n,m))\) \(\land\) eq.size E A \(\land\) length \(\eta\) = \(m\).

Finally, the real-valued and floating-point functional models for the matrix-matrix product are defined using the following polymorphic function, MM, which utilizes the matrix-vector product.

Variable dot : vector T \rightarrow vector T \rightarrow T.
Variables (A B : matrix T).

Definition MM : matrix \(\rightarrow\) matrix T := map (fun b \(\rightarrow\) MV dot A b) B.

E. Extension to MathComp

The correctness of the basic linear algebra operations defined above is supported by formal proofs connecting the real-valued operations VecAdd, DOT, MV, MatAdd, and MM to their counterparts in the Mathematical Components (MathComp) Library [12]. We use the mappings from Coq lists to MathComp matrices and vectors over the reals from Cohen et al [13] to prove that there is an injection from the LAProof functional models for these core operations to their corresponding MathComp operations. This mapping is particularly useful in two cases. Firstly, composing LAProof mixed backward-forward error bounds for matrix-matrix operations requires utilizing the ring properties of matrices, and MathComp provides extensive support for automatic rewriting over ring and field structures. Secondly, the big operators [14] in MathComp enable intuitive definitions of induced norms for normwise forward error bounds, which can be derived from our mixed backward-forward error bounds. Our proofs of correctness of the LAProof operations with respect to the MathComp operations allows clients of LAProof to lift the error bounds derived from the functional models over Coq lists to theorems over MathComp matrices and vectors.

In the following section, we illustrate how the floating-point functional models for the basic linear algebra operations introduced here can be connected to concrete C implementations, thereby guaranteeing the accuracy of practically useful programs.

IV. AN ACCURATE AND CORRECT C PROGRAM

In the previous section we described a functional model of the floating-point matrix-vector product (MVF) and introduced the formal proof in Coq of its accuracy. In this section, we describe a C program implementing compressed sparse row (CSR) matrix-vector multiplication, and a formal proof in Coq that this program exactly implements the floating-point functional model. We then compose the accuracy and correctness proofs in Coq to demonstrate that the C program is correct and accurate.

A. Sparse matrix-vector product

Compressed sparse row (CSR) is a standard data structure for sparse matrices that enables fast matrix-vector multiplications [15, §4.3.1]. The CSR format stores the elements of a sparse \(m \times n\) matrix \(A\) using three one-dimensional arrays: a floating-point array \(\text{val}\) that stores the nonzero elements of \(A\), an integer array \(\text{col}\_\text{ind}\) that stores the column indices of the elements in \(\text{col}\_\text{ind}\), and an integer array \(\text{row}\_\text{ptr}\) that stores the locations in the array \(\text{col}\_\text{ind}\) that start a row in \(A\). Figure 1 shows an example (from [15], adjusted for 0-based array indexing).

Our C implementation utilizes the CSR data structure given in Listing 1. The C function implementing sparse matrix-vector multiplication is shown in Listing 2. We use zero-based
void csr_mv_multiply (struct csr_matrix *m,
    double *v, double *p) {
unsigned i, rows = m->rows;
double *val = m->val;
unsigned *col_ind = m->col_ind;
unsigned *row_ptr = m->row_ptr;
unsigned next = row_ptr[0];
for (i = 0; i < rows; i++) {
double s = 0.0;
unsigned h = next;
next = row_ptr[i+1];
for (h = 0; h < next; h++) {
double x = val[h];
unsigned j = col_ind[h];
double y = v[j];
s = fma(x, y, s);
}
p[i] = s;
}
}

Listing 2. CSR matrix-vector multiplication in C.

B. Verifying the C program

We use VST [16] to specify and verify the C implementation of sparse matrix-vector multiplication. VST is a higher-order imperative logic for the C language embedded in the Coq proof assistant. As a variant of Hoare logics, judgements in VST take the familiar form of the Hoare triple \{Pre\} c \{Post\}, where the preconditions Pre and postcondition Post are assertions on program states. These assertions (preconditions, postconditions, loop invariants, etc.) are written as \textit{PROP}(P) \textit{LOCAL}(L) \textit{SEP}(R) where P is a sequence of Coq terms of type Prop, L characterizes the values of local and global variables, and R is a spatial assertion describing the contents of the heap. In function preconditions, in place of the \textit{LOCAL}() part we write \textit{PARAMS}(); and in function postconditions, we write \textit{RETURN}().

The VST specification of the CSR function (\textit{csr\_mv\_spec}) is shown in Listing 3. The specification and its corresponding proof show that the CSR function calculates the same floating-point computation as the dense matrix-multiply functional model, \textit{except} that where \(A_{ij} = 0\), the dense algorithm computes \(A_{ij} \cdot x_i + s\) where the sparse algorithm just uses \(s\). This is a notable difference in floating-point arithmetic, where it is not always the case that \(0 \cdot y + s = s\), for instance when \(y = \infty\) or NaN. Even when \(y\) and \(s\) are finite values in the format, it is not always true that \(y \cdot 0 + s\) is the same floating-point value as \(s\) because of signed zeros. Finally, even when all elements of the matrix \(A\) and vector \(x\) are finite, we cannot assume that the intermediate results \(s\) are finite as the computation may introduce overflow.

Thus, when specifying the correctness of matrix-vector multiplication we must tread carefully: we reason modulo equivalence relations. We define \textit{eq} \(x\) \(y\) to mean that either both \(x\) and \(y\) are finite and equal (with \(+0 = -0\)), or neither is finite (both are infinities or NaNs). Our function will have a precondition that \(A\) and \(x\) are all finite, and postcondition that the computed result is \textit{eq} \(x\) \(y\) to the result that a dense matrix multiply algorithm would compute. For such reasoning we use Coq’s \textit{Parametric Morphism} system for reasoning over partial equivalence relations using rewrite rules [17].

The \textit{WITH} in the CSR specification \textit{csr\_mv\_spec} quantifies over 8 logical variables that appear in both the precondition and the postcondition. The variable \(A\) is the formal model of the floating-point matrix, and \(x\) is the model of the vector. Pointer value \(m\) is the address of a CSR representation of \(A\), and \(v\) is the address of the array containing values \(x\). \(\pi_1, \pi_2\) are permission-shares for read access to \(A\) and \(x\), and \(\pi_3\) specifies write permission for address \(p\) where the output vector \(y\) is to be stored.

The precondition \textit{PRE} in \textit{csr\_mv\_spec} asserts that, given 3 parameters whose C-language types are (respectively) pointer-to-struct, pointer-to-double, pointer-to-double;

- \textit{PROP}: the input arrays are readable and the output array is writable; every row of the matrix has the same length as vector \(x\); the dimensions of \(A\) and \(x\) are representable as C integers; all the values in \(A\) and \(x\) are finite;
- \textit{PARAMS}: the values of the function parameters are the values \(m, v, p\), respectively; and
- \textit{SEP}: the data structures in memory represent \(A\) at address
In writing the precondition, we use an abstract data type
representation relation $\text{csr\_rep}$ to describe the data stored at
address $m$.

The postcondition $\text{POST\_rep}$ asserts the following: there exists a
float-vector $y$ that is equivalent to the floating-point product
$Ax$; this result is stored at address $p$; and the data at $m$ and $v$ is
undisturbed. Furthermore, VST’s program logic guarantees that
any data not mentioned in the SEP clauses remains undisturbed.

The user-defined representation relation $\text{csr\_rep}$. $A \ m$ says
that matrix $A$ is represented as a data structure at address $m$.
In turn it relies on a functional model of sparse matrices. We
define this functional model at any floating-point type $t$
(single-precision, double-precision, half, quad, etc.). We prove
lemmas in Coq about the representation relation, and use
those to prove in VST (embedded in Coq) that the C function
satisfies the $\text{csr\_mv\_spec}$.

V. RELATED WORK

The challenge of finding practically useful methods for
guaranteeing the correctness and accuracy of numerical pro-
grams is an old one. While a variety of approaches have been
successfully explored, formal static analysis methods have
historically been the least prominent. We concentrate here on
the closest related work, which we believe falls into three
fairly distinct categories: formal tools for floating-point error
analysis, formalizations of numerical linear algebra, and end-
to-end machine checked proofs.

Formal tools for floating-point error analysis: There are
several tools that perform rounding error analysis and generate
machine-checkable proof certificates with varying levels of
automation: Gappa [18] is implemented in C++ and produces
proof scripts that can be checked in Coq; PRECiSA [19] is
implemented in Haskell and C and generates proofs in the
PVS [20] proof assistant, FPTaylor [21] is implemented in
OCaml and can produce proof certificates in HOL Light;
VCFloat [10], [11] is implemented in Coq; and Daisy [22] is
implemented in Scala, producing proof scripts that can be
checked by Coq and HOL4 [23]. In general, these tools focus
on automatically obtaining tight forward error bounds for
arithmetic expressions in a given precision—that is, straight-
line loop bodies. The goal of the LAProof library is funda-
mentally different: to provide formal proofs of widely accepted
mixed forward-backward error bounds for standard algorithms
that can be used modularly in larger verification efforts.

Formalizations of numerical linear algebra: With regard
to the basic linear algebra operations, Roux [24] formally
proved forward error bounds in Coq for finite precision inner
product and summation, and used these bounds to derive a
formal mechanized proof of the accuracy of a finite precision
algorithm for the Cholesky decomposition. The author proves
that the formal model for floating-point arithmetic used in their
formalization satisfies the IEEE 754 binary format specified by
Flocq.

End-to-end machine-checked proofs: We demonstrated
the intended functionality of the LAProof library with the
verification of a C program implementing sparse matrix-
vector multiplication. Rather than serving as its own end-
to-end verification effort, the LAProof library is intended to
serve as a proof layer between the verification of application
software and programs implementing operations defined by
BLAS. There are a few end-to-end machine-checked proofs of
numerical programs in the literature that we believe could have
benefited from modular, verified building blocks like those
provided by LAProof.

Boldo and co-authors developed a machine-checked Coq
proof of the correctness and accuracy of a C program imple-
menting a second-order finite difference scheme for solving
the one-dimensional acoustic wave equation [25]. Scaling their
results to higher dimensions would require formal error bounds
for the accuracy of basic linear algebra operations. Similarly,
Kellison and co-authors developed a machine checked proof of
the correctness and accuracy of an implementation of velocity-
Verlet integration of the simple harmonic oscillator [26]. They
obtain a forward error bound for the round off error of their
method, but a mixed backward-forward error result for matrix-
vector multiplication could have produced a tighter and more
general bound.

VI. CONCLUSION

The LAProof library provides a promising modular proof
layer between the verification of application software and the
verification of programs implementing linear algebra opera-
tions defined by the BLAS standard. The formal roundoff
error analysis provided provided by LAProof carefully handles
underflow and overflow, and captures all higher-order error
terms. We have demonstrated a practical case study of how
LAProof can be used as such an interface by connecting the
LAProof implementation of matrix-vector product, for which
LAProof provides a formally guaranteed error bound, to a
C program implementing sparse matrix-vector multiplication
using the compressed sparse row format.

A natural question arises concerning the ease of using the
LAProof library in verification efforts other than the sparse
matrix-vector multiplication example we have described. We
believe that we have made at least two design choices that will
support the porting of LAProof to other verification efforts.

Firstly, rather than using the Mathematical Components
Library [12] directly to define our functional models in Coq, we
chose to implement our functional models using Coq’s
standard lists over arbitrary types. This ensures that LAProof
is a middle ground between the verification of programs using
tools like VST, which tend to use concrete Coq types, and the
abstract and dependent types used by the MathComp library,
which are more useful when proving abstract properties of
programs. Our proofs of correctness of the LAProof opera-
tions with respect to MathComp operations over matrices
and vectors allows clients of LAProof to lift the error bounds
derived from the LAProof functional models over Coq lists to
theorems using MathComp.
Secondly, mixed backward-forward error bounds separate rounding errors from the stability of the mathematical problem being solved by the application software more clearly than forward error bounds. The roundoff error analysis provided by LAPoof should therefore be more widely usable than forward error bounds alone.

A limitation of providing mixed backward forward error bounds to clients of LAPoof is automation. Forward error analysis requires successively accumulating the error introduced by each floating-point operation, while backward error analysis produces bounds of the form of equation (3), which requires identifying the error terms generated by each operation that can be propagated back onto the operands. In comparison to forward error, automated backward error analysis for numerical software has received limited treatment in the literature [27].

We conclude by noting that while the roundoff error analyses in LAPoof are performed for particular implementations in Coq, the formal statement of LAPoof theorems can serve as an interface to which other implementations can be shown to adhere. It is our hope that LAPoof can therefore serve as a proof interface with a reference implementation – in the spirit of BLAS – in the formal verification of numerical programs.

VII. ACKNOWLEDGMENTS

This research was supported in part by NSF Grants CCF-2219997 and CCF-2219757, by a U.S. Department of Energy Computational Science Fellowship DE-SC0021110, and by the Chateaubriand fellowship program.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES


