LAProof: a library of formal proofs of accuracy and correctness for linear algebra programs

Ariel E. Kellison
Dept. of Computer Science
Cornell University
Ithaca, NY, USA
0000-0003-3177-7958

Andrew W. Appel
Dept. of Computer Science
Princeton University
Princeton, NJ, USA
0000-0001-6009-0325

Mohit Tekriwal
Dept. of Aerospace Engineering
University of Michigan
Ann Arbor, MI, USA
0000-0002-5103-2630

David Bindel
Dept. of Computer Science
Cornell University
Ithaca, NY, USA
0000-0002-8733-5799

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 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 compressed-row-storage (CRS) sparse matrix-vector multiplication. 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 Flcoup [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 754 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 demonstrate verification of a C program implementing sparse matrix-vector multiply using the compressed sparse row format (CSR).

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 754 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 compressed sparse row (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-analysis [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 Flocc library [3], and are derived using the standard rounding error model for floating-point arithmetic [6, sect 2.2]:

\[
\text{fl}(a \text{ op } b) = (a \text{ op } b)(1 + \delta) + \epsilon
\]

(1)

where \( \delta, \epsilon = 0 \) for op \( \in \{ +, -, \times, /, \sqrt{\cdot} \} \),

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 \( \phi_R(x) \) defined over Coq’s axiomatic real numbers that represents the operation in exact arithmetic, and a floating point valued function \( \phi_{F,e}(x) \) defined over the IEEE 754 format specified by Flocc. Using these functional programs, the absolute forward error \( F \) is expressed as

\[
F \triangleq |\phi_R(x) - \phi_{F,e}(x)|.
\]

(2)

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

\[
\phi_{F,e}(x) = \phi_R(x + \Delta x) + \bar{\delta}.
\]

(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, 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 ( r \leftarrow x \cdot y )</td>
</tr>
<tr>
<td>sVec ( r \leftarrow \alpha x )</td>
</tr>
<tr>
<td>SUM ( r \leftarrow \sum_i x_i )</td>
</tr>
<tr>
<td>VecAdd ( r \leftarrow x + y )</td>
</tr>
<tr>
<td>VecAXPBY ( r \leftarrow \alpha x + \beta y )</td>
</tr>
<tr>
<td>VecNRM1 ( r \leftarrow |x|_1 )</td>
</tr>
<tr>
<td>VecNRM2 ( r \leftarrow |x|_2 )</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 (bDOT).** For any two vectors \( \mathbf{u}, \mathbf{v} \in \mathbb{F}^{n,e} \), the vectors \( \mathbf{\hat{u}}, \mathbf{\hat{v}} \in \mathbb{R}^n \) and real number \( c \in \mathbb{R} \) exist such that

\[
\text{fl}(\mathbf{u} \cdot \mathbf{v}) = \mathbf{\hat{u}} \cdot \mathbf{\hat{v}} + c,
\]

(6)

where \( |c| \leq g(n, n) \) and every \( k \)th element of \( \mathbf{\hat{u}} \) 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 \( \mathbf{u} \) and \( \mathbf{v} \) 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 \( |\delta_k| \leq u, |\gamma_k| \leq u, \) 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

\[
\tilde{\gamma}_j = \prod_{\ell=j}^{n} (1 + \gamma_\ell) - 1, \quad \tilde{\delta}_j = (1 + \delta_j)(1 + \tilde{\gamma}_j) - 1
\]

for which we have the bounds

\[
|\tilde{\gamma}_j| \leq h(n-j) \leq h(n-1), \quad |\tilde{\delta}_j| \leq h(n-j+1) \leq h(n).
\]

The computed dot product is

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

or, equivalently

\[
\hat{s}_n = \mathbf{\hat{u}} \cdot \mathbf{\hat{v}} + c
\]
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]. Assuming the absence of overflow, LAProof provides the following error bound for matrix-vector multiplication.

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

\[
\tilde{\mathbf{f}}(\mathbf{M}\mathbf{v}) = (\mathbf{M} + \Delta \mathbf{M})\mathbf{u} + \eta,
\]

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 \(\Delta \mathbf{M}\) respects the bound \(|\Delta \mathbf{M}| \leq h(n)|\mathbf{M}|\).

---

**TABLE II**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Error Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>MV</td>
<td>(r \leftarrow Ax)</td>
</tr>
<tr>
<td>sMV</td>
<td>(r \leftarrow \alpha Ax)</td>
</tr>
<tr>
<td>GEMV</td>
<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 \(\mathbf{u} \in \mathbb{F}_p^n\) and matrix \(\mathbf{M} \in \mathbb{F}_p^{m \times n}\), if the elements of \(\mathbf{u}\) and \(\mathbf{M}\) are 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 \(g(n+1, n) \leq 2^e\), then the floating-point result of \(\tilde{\mathbf{f}}(\mathbf{M}\mathbf{u})\) is a vector with elements that are finite numbers in the floating point format \(\mathbb{F}_p\).

**C. Matrix Operations**

The core matrix operations in LAProof are the matrix-matrix product (\(\mathbf{R} \leftarrow \mathbf{AB}\)), matrix addition (\(\mathbf{R} \leftarrow \mathbf{A} + \mathbf{B}\)), and scaling by a constant (\(\mathbf{R} \leftarrow \alpha \mathbf{A}\)). 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 (\(\mathbf{R} \leftarrow \alpha \mathbf{AB}\)), the addition of a scaled matrices (\(\mathbf{R} \leftarrow \alpha \mathbf{A} + \beta \mathbf{Y}\)), and a scaled matrix-matrix product plus a scaled matrix (\(\mathbf{R} \leftarrow \alpha \mathbf{AX} + \beta \mathbf{Y}\)).
We define a floating point functional model D0TF by supplying D0T with a generic ftype(t) type and the appropriate functions over this type. LAProof uses the VCFloat functions BPLUS : ftype t → ftype t → ftype t and BMULT : ftype t → ftype t → ftype t for addition and multiplication over ftype t. These VCFloat functions are simply wrappers around the corresponding IEEE 754 operators defined in Flocq, so LAProof inherits (and extensively utilizes) the soundness of these operators with respect to the IEEE 754 specification formalized by Flocq. A real-valued functional model D0TR 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 R) (\eta : R),
ETA2 (DOTF u v) = D0TR u' (map FT2R v) + \eta
\land (\forall i, (i < n) \rightarrow \exists \delta, u'_i = (1 + \delta) FT2R (v_i)
\land |\delta| \leq b(n) \land |\eta| \leq g(n,n)),

where the function FT2R : ftype t → R is the 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,

Variable add mul : T → T → T.

Definition VecAdd (u v : vector T): vector T
:= map (uncurry add) (combine u v).

Definition SUM: vector 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 D0TF) to a polymorphic function MV defined over an arbitrary implicit element type T:

Variable 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.

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 add).

Denoting real-valued MatAdd and real-valued VecAdd as \( +_m \) and \( +_e \), respectively, and floating-point valued and real-valued \( MV \) as \( \otimes \), and \( *_{ MV } \), 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 \( Ar := (\text{map} \ (\text{map} \ \text{FT2R}) \ A) \).
Notation \( vr := (\text{map} \ \text{FT2R} \ v) \).

Hypothesis Hfin : \( \exists \eta \in \text{finite}_\text{vec} \ (\text{MV} \ A \ v) \).

Hypothesis Hlen : \( \forall x, \text{In} \ x \ A \rightarrow \text{length} \ x = n \).

Theorem bfMV: \( \exists \ (E : \text{matrix} \ (\text{ftype} \ t)) \). \( \text{map} \ \text{FT2R} \ (A \otimes v) = (Ar +_m E) *_{ MV } vr +_e \eta \)
\( \wedge (\forall \ i,j, (i < m) \rightarrow (j < n) \rightarrow |E_{ij}| \leq h(n)|Ar_{ij}|) \)
\( \wedge (\forall k, \text{In} k \ \eta \rightarrow |k| \leq g(n,n)) \)
\( \wedge \text{eq}_\text{size} \ E \ A \wedge \text{length} \ \eta = m \).

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

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

Definition \( \text{MM} : \text{matrix} T \)
:= \( \text{map} \ (\text{fun} \ b \Rightarrow \text{MV} \ \text{dot} \ A \ b) \cdot B \).

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 [?] 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 (\( \text{MV} \)) 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 [14, §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 [14], 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 indexing for arrays and matrices. The \( i \)th element of \( \text{row}_\text{ptr} \) points into an offset within \( \text{val} \) and within \( \text{col}_\text{ind} \) where the \( i \)th row is represented. For \( \text{row}_\text{ptr}(i) \leq h < \text{row}_\text{ptr}(i+1) \),
\textbf{Definition} \texttt{csr\_mv\_spec :=}

\texttt{DECLARE \_csr\_mv\_multiply}

\texttt{WITH \_1: share, \_2: share, \_3: share,}

\texttt{m: val, A: TDouble, v: val,}

\texttt{x: vector TDouble, p: val}

\texttt{PRE [ tptr t\_csr, tptr tdouble, tptr tdouble ]}

\texttt{PROP (readable\_share \_1; readable\_share \_2;}

\texttt{writable\_share \_3;}

\texttt{matrix\_cols A (Zlength x);}

\texttt{matrix\_rows A < Int\_max\_unsigned;}

\texttt{Zlength x < Int\_max\_unsigned;}

\texttt{Forall finite x;}

\texttt{Forall (Forall finite A)}

\texttt{PARAMS (m; v; p)}

\texttt{SEP (\_1 A m;}

\texttt{data\_at \_2 (tarray tdouble (Zlength x))}

\texttt{(map Vfloat x) v;}

\texttt{data\_at \_3 (tarray tdouble (matrix\_rows A)) p)}

\texttt{POST [ tvoid ]}

\texttt{EX y: vector TDouble,}

\texttt{PROP (forall2 \_3 \_2 \_1 \_1 (MVF A x))}

\texttt{RETURN()}

\texttt{SEP (\_1 A m;}

\texttt{data\_at \_2 (tarray tdouble (Zlength x))}

\texttt{(map Vfloat x) v;}

\texttt{data\_at \_3 (tarray tdouble (matrix\_rows A))}

\texttt{(map Vfloat y) p)}

\texttt{Listing 3. Function specification for CSR matrix-vector multiply}

Thus, when specifying the correctness of matrix-vector multiplication we must tread carefully: we reason modulo equivalence relations. We define \texttt{feq} \(x\) to \(y\) 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 \texttt{feq} to the result that a dense matrix multiply algorithm would compute. For such reasoning we make heavy use of Coq’s \texttt{Parametric Morphism} system for reasoning over partial equivalence relations using rewrite rules [16].

The \texttt{WITH} in the CSR specification \texttt{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 \(x\). \(\_1, \_2, \_3\) are permission-shares for read access to \(A\) and \(x\), and \(\_3\) specifies write permission for address \(p\) where the output vector \(y\) is to be stored.

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

- \texttt{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;
- \texttt{PARAMS}: the values of the function parameters are the values \(m, v,\) and \(p\), respectively; and

\begin{verbatim}
struct csr\_matrix {
  double *val;
  unsigned *col\_ind, *row\_ptr, rows, cols;
};

Listing 1. A CSR struct in C.

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.

\end{verbatim}
• SEP: the data structures in memory represent $A$ at address $m$, and $x$ at address $v$, and address $p$ has an uninitialized array (to hold the result).

In writing the precondition, we use an abstract data type representation relation $csr_{\text{rep}}$ to describe the data stored at address $m$.

The postcondition $\text{POST}$ 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 $csr_{\text{rep}} A m$ 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 $csr_{\text{mv}}_{\text{spec}}$.

V. RELATED WORK

The challenge of finding practically useful methods for guaranteeing the correctness and accuracy of numerical programs 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 [17] is implemented in C++ and produces proof scripts that can be checked in Coq; PRECISA [18] is implemented in Haskell and C and generates proofs in the PVS [19] proof assistant, FPTaylor [20] is implemented in OCaml and can produce proof certificates in HO Light; VCFloat [10], [11] is implemented in Coq; and Daisy [21] is implemented in Scala and can produce proof scripts that can be checked by both Coq and HOL4 [22]. 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 fundamentally 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 [23] formally proved in Coq forward error bounds for finite precision inner product and summation, and used these bounds to provide a formal use 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 Floq.

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 implementing a second-order finite difference scheme for solving the one-dimensional acoustic wave equation [24]. 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 [25]. 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 operations 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 concrete implementation of 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 operations with respect to MathComp operations over matrices and vectors allows clients of LAProof to lift the error bounds.
derived from the LAProofs 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 LAProof should therefore be more widely usable than forward error bounds alone.

A limitation of focusing on providing mixed backward forward error bounds to clients of LAProof 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 inputs of the function. Automatically performing backward error analysis is a challenge which hasn’t been addressed as completely in the literature as forward error analysis [2].

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

REFERENCES


