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

April 29, 2023

***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 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 $\mathcal{O}(u^2)$.**

***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 δ_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 $nu+\mathcal{O}(u^2)$. 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 Flocq [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 Flocq library [3], and are derived using the standard rounding error model for floating-point arithmetic [6, sect 2.2]:

$$\begin{aligned} \mathrm{fl}(a\ op\ b) &= (a\ op\ b)(1+\delta)+\epsilon \qquad (1)\\ |\delta| \le u,\ |\epsilon| &\le \eta,\ \delta\epsilon = 0,\ op \in \{+,-,\times,/,\surd\}, \end{aligned}$$

where $\delta, \epsilon = 0$ for $op \in \{+,-\}$ [7, Theorem 1]. 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 $\eta = 2^{3-e-p-1}$, and the exactly representable numbers in the format by $\mathbb{F}_{p,e}$.

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

$$F \triangleq |\phi_{\mathbb{R}}(x) - \phi_{\mathbb{F}_{p,e}}(x)|. \qquad (2)$$

The mixed backward-forward error requires deriving a suitable perturbation (Δx) to the inputs of $\phi_{\mathbb{R}}$ and small forward error term $(\hat{\delta})$ such that

$$\phi_{\mathbb{F}_{p,e}}(x) = \phi_{\mathbb{R}}(x+\Delta x) + \hat{\delta}. \qquad (3)$$

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:

$$h(n) = ((1+u)^n - 1). \qquad (4)$$
$$g(n,m) = n\eta(1+h(m)). \qquad (5)$$

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
LAProof Vector Operations

DOT	$r \leftarrow x \cdot y$
sVec	$r \leftarrow \alpha x$
SUM	$r \leftarrow \sum_i x_i$
VecAdd	$r \leftarrow x + y$
VecAXPBY	$r \leftarrow \alpha x + \beta y$
VecNRM1	$r \leftarrow \|x\|_1$
VecNRM2	$r \leftarrow \|x\|_2$

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 $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^n_{p,e}$, the vectors $\hat{\boldsymbol{u}}, \boldsymbol{\delta} \in \mathbb{R}^n$ and real number $c \in \mathbb{R}$ exist such that*

$$\mathrm{fl}(\boldsymbol{u}\cdot\boldsymbol{v}) = \hat{\boldsymbol{u}}\cdot\boldsymbol{v} + c, \qquad (6)$$

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

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 $\tilde{s}_1 = u_1 v_1(1+\delta_1) + \epsilon_1$ for $k = 1$, and

$$\tilde{s}_k = (\tilde{s}_{k-1} + u_k v_k(1+\delta_k) + \epsilon_k)(1+\gamma_k) \qquad (7)$$

with $|\delta_k| \le u$, $|\gamma_k| \le u$, and $|\epsilon_k| \le \eta$. If we define $\gamma_1 = 0$, we have

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

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

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

The computed dot product is

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

or, equivalently

$$\tilde{s}_n = \hat{\mathbf{u}}\cdot\mathbf{v} \ + c$$

where $|c| \le g(n,n)$ and $\hat{u}_j = u_j(1+\tilde{\delta}_j)$. 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

$$|\,\mathrm{fl}(\mathbf{u}\cdot\mathbf{v}) - (\mathbf{u}\cdot\mathbf{v})| \le S(\mathbf{u},\mathbf{v})h(n) + h(n,n-1) \tag{8}$$

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 $\mathbf{u},\mathbf{v} \in \mathbb{F}^n_{p,e}$ is sparse, we prove the theorem **sbfDOT**, in which the error functions g and h in theorem **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 $\mathbf{u},\mathbf{v} \in \mathbb{F}^n_{p,e}$, the floating-point partial sums in equation 7 become

$$\tilde{s_k} = \mathrm{FMA}(u_k, v_k, \tilde{s}_{k-1}) = (u_k v_k + \tilde{s}_{k-1})(1+\delta_k) + \epsilon_k$$

where $|\delta_k| \le u$ and $|\epsilon_k| \le \eta$. 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 of elements in a vector, the LAProof library provides the following backward error bound.

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

$$\mathrm{fl}\left(\sum_{i=1}^{n} u_i\right) = \sum_{i=1}^{n} \hat{u}_i, \tag{10}$$

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

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 it 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 $\boldsymbol{u} \in \mathbb{F}^n_{p,e}$, and matrix $\boldsymbol{M} \in \mathbb{F}^{m\times n}_{p,e}$, there exist a matrix $\boldsymbol{\Delta M} \in \mathbb{R}^{m\times n}$ and vector $\boldsymbol{\eta} \in \mathbb{R}^n$ such that*

$$\mathrm{fl}(\boldsymbol{Mv}) = (\boldsymbol{M} + \boldsymbol{\Delta M})\boldsymbol{u} + \boldsymbol{\eta}, \tag{11}$$

where every element of the backward error term $\boldsymbol{\eta}$ respects the bound $|\eta| \le g(n,n)$ and each element of the forward error term $\boldsymbol{\Delta M}$ respects the bound $|\Delta M| \le h(n)|M|$.

TABLE II
LAPROOF MATRIX-VECTOR OPERATIONS

MV	$r \leftarrow Ax$
sMV	$r \leftarrow \alpha Ax$
GEMV	$r \leftarrow \alpha Ax + \beta y$

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 $\boldsymbol{u} \in \mathbb{F}^n_{p,e}$, and matrix $\boldsymbol{M} \in \mathbb{F}^{m\times n}_{p,e}$, if the elements of $\boldsymbol{u}$ and $\boldsymbol{M}$ are bounded by the square root of*

$$\left(\frac{2^e - \eta}{1+u} - g(n,n-1)\right)\left(\frac{1}{1+n(1+u)^n}\right) \tag{12}$$

and $g(n+1,n) \le 2^e$, then the floating-point result of $\mathrm{fl}(\boldsymbol{Mu})$ is a vector with elements that are finite numbers in the floating point format $\mathbb{F}_{p,e}$.

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 ($R \leftarrow \alpha AX + \beta Y$).

TABLE III
LAPROOF MATRIX OPERATIONS

sMat	$R \leftarrow \alpha A$
MM	$R \leftarrow AB$
MatAdd	$R \leftarrow A + B$
sMM	$R \leftarrow \alpha AB$
MatAXPBY	$R \leftarrow \alpha X + \beta Y$
GEMM	$R \leftarrow \alpha AX + \beta Y$

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 t`, 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 Flocq formalization of IEEE-754 binary floating-point formats. We begin our description of the LAProof functional models by first providing 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 $\mathbb{F}^n_{p,e}$ 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 (dot) 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 are 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 : 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 `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: ∃ (u' : list R) (η : R),
  FT2R (DOTF u v) = DOTR u' (map FT2R v) + η
  ∧ (∀ i, (i < n) → ∃ δ, u'_i  = (1 + δ) FT2R (v_i)
  ∧   |δ| ≤ h(n)  )   ∧ |η| ≤ g(n,n)),
```

where the function `FT2R : ftype t` $\rightarrow \mathbb{R}$ 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`,

```
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 → 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`:

```
Variable dot : vector T → vector T → T.
Variables (A : matrix T) (v : vector 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 $+_v$, respectively, and floating-point valued and real-valued MV as $\otimes_v$ and $*_v$, respectively, the formal LAProof statement of theorem 3 is given as follows.

```
Variable (A : matrix (ftype t)).
Variable (v : vector (ftype t)).
Let m := (length A).
Let n := (length v).
Notation Ar := (map (map FT2R) A).
Notation vr := (map FT2R v).
Hypothesis Hfin : is_finite_vec (MVF A v).
Hypothesis Hlen : ∀ x, In x A → length x = n.

Theorem bfMV: ∃ (E : matrix R) (η : vector R),
  map FT2R (A ⊗_v v) =  (Ar +_m E) *_v vr +_v η
  ∧ (∀ ij, (i < m) → (j < n) → |E_{i,j}| ≤ h(n)|Ar_{i,j}|
  ∧ (∀ k, In k η → |k| ≤ g(n, n))
  ∧ eq_size E A  ∧ length η = m.
```

Finally, the real-valed 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 → vector T → T.
Variables (A B : matrix T).

Definition MM : matrix T
    := map (fun b ⇒ 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 [?] 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.

$$\begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}$$

val	10	−2	3	9	3	7	8	7	3 · · · 9	13	4	2	−1
col_ind	0	4	0	1	5	1	2	3	0 · · · 4	5	1	4	5
	0		2			5			8	16			19

row_ptr	0	2	5	8	12	16	19

Fig. 1. An example of the three arrays (val, col_ind, row_ptr) used to store a matrix compressed sparse row (CSR) format.

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 [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 val that stores the nonzero elements of A, an integer array col_ind that stores the column indices of the elements in col_ind, and an integer array row_ptr that stores the locations in the array col_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 ith element of row_ptr points into an offset within val and within col_ind where the ith row is represented. For row_ptr$(i) \leq h <$ row_ptr$(i+1)$,

```
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 = row4_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.

```
Definition csr_mv_spec :=
DECLARE _csr_mv_multiply
WITH π1: share, π2: share, π3: share,
      m: val, A: matrix Tdouble, v: val,
      x: vector Tdouble, p: val
PRE [ tptr t_csr, tptr tdouble, tptr tdouble ]
  PROP(readable_share π1; readable_share π2;
         writable_share π3;
         matrix_cols A (Zlength x);
         matrix_rows A < Int.max_unsigned;
         Zlength x < Int.max_unsigned;
         Forall finite x;
         Forall (Forall finite) A)
  PARAMS(m; v; p)
  SEP (csr_rep π1 A m;
       data_at π2 (tarray tdouble (Zlength x))
                                (map Vfloat x) v;
       data_at_ π3 (tarray tdouble (matrix_rows A
    )) p)
POST [ tvoid ]
 EX y: vector Tdouble,
  PROP(Forall2 feq y (MVF A x))
  RETURN()
  SEP (csr_rep π1 A m;
       data_at π2 (tarray tdouble (Zlength x))
                                (map Vfloat x) v;
       data_at π3 (tarray tdouble (matrix_rows A))
                                (map Vfloat y) p).
```

Listing 3. Function specification for CSR matrix-vector multiply

col_ind$(h) = j$, we have $A_{ij} =$ val(h); and elsewhere $A_{ij} = 0$.

B. Verifying the C program

We use VST [15] to specify and verify the C implementation of sparse matrix-vector multiplication. VST is a higher-order impredicative 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. VST operates on assertions of the form `PROP(P) LOCAL(Q) SEP(R)` where `P` is a Coq term of type `Prop`, `L` characterizes the values of local and global variables, and `R` is a spatial assertion describing the contents of the heap.

The VST specification of the CSR function (`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, *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 is ∞ 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 computatation may introduce overflow.

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

The `WITH` in the CSR specification `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. π_1, π_2 are permission-shares for read access to A and x, and π_3 specifies write permission for address p where the output vector y is to be stored.

The precondition `PRE` in `csr_mv_spec` asserts that, given 3 parameters whose C-language types are (respectively) pointer-to-struct, pointer-to-double, pointer-to-double;

- `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;
- `PARAMS`: the values of the function parameters are the values m, v, and p, respectively; and

- `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_rep` to describe the data stored at address m.

The postcondition `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 guaratees that any data not mentioned in the `SEP` clauses remains undisturbed.

The user-defined *representation relation* `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 `csr_mv_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 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 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 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 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 [**?**].

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

[1] "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, p. 135–151, Jun. 2002. [Online]. Available: https://doi.org/10.1145/567806.567807

[2] "Basic linear algebra subprograms technical (BLAST) forum standard," *International Journal of High Performance Computing Applications: Special Issue - Part I & II*, vol. 16, no. 1-2, pp. 1–199, 2002-01 2002.

[3] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in Coq," in *2011 IEEE 20th Symposium on Computer Arithmetic*, 2011, pp. 243–252.

[4] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[5] J. W. Demmel, *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611971446

[6] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Society for Industrial and Applied Mathematics, 2002. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718027

[7] J. R. Hauser, "Handling floating-point exceptions in numeric programs," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 2, p. 139–174, Mar. 1996. [Online]. Available: https://doi.org/10.1145/227699.227701

[8] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo, "Design, implementation and testing of extended and mixed precision BLAS," *ACM Trans. Math. Softw.*, vol. 28, no. 2, p. 152–205, Jun. 2002. [Online]. Available: https://doi.org/10.1145/567806.567808

[9] G. W. Stewart, *Matrix Algorithms*. Society for Industrial and Applied Mathematics, 1998. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611971408

[10] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin, "A unified coq framework for verifying c programs with floating-point computations," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, ser. CPP 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 15–26. [Online]. Available: https://doi.org/10.1145/2854065.2854066

[11] A. W. Appel and A. E. Kellison, "VCFloat2: Floating-point error analysis in Coq," 2022. [Online]. Available: https://github.com/VeriNum/vcfloat/blob/master/doc/vcfloat2.pdf

[12] A. Mahboubi and E. Tassi, *Mathematical Components*. Zenodo, Sep. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7118596

[13] J. M. Cohen, Q. Wang, and A. W. Appel, "Verified erasure correction in Coq with MathComp and VST," in *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part II*, Berlin, Heidelberg, 2022, p. 272–292. [Online]. Available: https://doi.org/10.1007/978-3-031-13188-2_14

[14] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[15] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel, "VST-Floyd: A separation logic tool to verify correctness of C programs," *J. Autom. Reason.*, vol. 61, no. 1-4, pp. 367–422, Jun. 2018.

[16] M. Sozeau, "A new look at generalized rewriting in type theory," in *1st Coq Workshop Proceedings*, H. Herbelin, Ed. Technische Universitaet Muenchen, Aug. 2009.

[17] S. Boldo, J.-C. Filliâtre, and G. Melquiond, "Combining Coq and Gappa for certifying floating-point programs," in *International Conference on Intelligent Computer Mathematics*. Springer, 2009, pp. 59–74.

[18] M. M. Moscato, L. Titolo, A. Dutle, and C. A. Muñoz, "Automatic estimation of verified floating-point round-off errors via static analysis," in *Computer Safety, Reliability, and Security - 36th International Conference, SAFECOMP'17*, 2017, pp. 213–229.

[19] S. Owre, J. M. Rushby, and N. Shankar, "Pvs: A prototype verification system," in *Automated Deduction—CADE-11*, D. Kapur, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 748–752.

[20] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions," *ACM Transactions on Programming Languages and Systems*, vol. 41, no. 1, pp. 1–39, 2018.

[21] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, "Daisy - framework for analysis and optimization of numerical programs (tool paper)," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 270–287.

[22] H. Becker, N. Zyuzin, R. Monat, E. Darulova, M. O. Myreen, and A. Fox, "A verified certificate checker for finite-precision error bounds in coq and hol4," in *2018 Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–10.

[23] P. Roux, "Formal Proofs of Rounding Error Bounds," *Journal of Automated Reasoning*, p. 23, 2015. [Online]. Available: https://hal.science/hal-01091189

[24] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis, "Trusting computations: A mechanized proof from partial differential equations to actual program," *Computers and Mathematics with Applications*, vol. 68, no. 3, pp. 325–352, 2014.

[25] A. E. Kellison and A. W. Appel, "Verified numerical methods for ordinary differential equations," in *15th Int'l Workshop on Numerical Software Verification*, 2022.