Starting in the late 1960s computer scientists including Dijkstra and Hoare advocated goal-oriented programming and the formal derivation of algorithms. The chief impediment to realizing this for loop-based programs was that a priori determination of loop-invariants, a prerequisite for developing loops, was a task too complex for any but the simplest of operations. Around 2000, these techniques were for the first time successfully applied to the domain of high-performance dense linear algebra libraries. This has led to a multitude of papers, mostly published in the ACM Transactions for Mathematical Software, as well as a system for the mechanical derivation of algorithms and a high-performance linear algebra library, \textit{libflame}, that includes more than a thousand variants of algorithms for more than a hundred linear algebra operations. To our knowledge, this success story has unfolded with limited awareness on the part the formal methods community. This paper reports on ten years of experience and is meant to raise that awareness.

1 Introduction

Linear algebra libraries reside at the bottom of the scientific computing food chain. While most practical applications give rise to sparse linear algebra problems, a significant number of them spends most computational time solving dense matrix problems. Even sparse linear algebra problems often have dense subproblems to be solved. As a result, LAPACK [1], a package for dense matrix operations, developed in the late 1980s and early 1990s, is undoubtedly the most commonly used library in this field.

Since 2000, the FLAME project at The University of Texas at Austin, Universidad Jaume I (Spain), and RWTH Aachen University (Germany) has been pursuing a modern replacement of LAPACK, \textit{libflame} [28]. The domain poses a few interesting challenges: scientific computing tends to exploit the latest architectures, demanding the highest possible performance. This requires the design of loop-based algorithms that cast most computation in terms of high-performing matrix-matrix operations (like matrix-matrix multiplication). The loop steps through matrices with block sizes chosen so as to optimize the reuse of data in caches. For a specific operation, there are often multiple algorithmic variants, with one algorithmic variant matching a given architecture better than the others, yielding higher performance. Thus, a library should incorporate multiple loop-based algorithms for a given operation so that the best one can be chosen. The LAPACK library does not include such a wide variety of algorithms. This brings up the question of how to systematically find algorithmic variants for a given operation. The formal derivation of loop-based algorithms turns out to be the answer [2, 3, 4, 17, 18, 19, 25, 26], as we will illustrate.

This paper does not provide a scholarly treatment of the field of derivation of algorithms. All we have ever needed to develop the described techniques is given in the text by Gries [16], which itself is based on the works of Dijkstra [7, 8] and Hoare [20]. What the paper does provide is what we believe to be an excellent practical example of the application of formal derivation of loops to the domain of dense linear algebra.
2 Derivation of Linear Algebra Algorithms

In this section, we walk the reader through the derivation process. The procedure is completely routine for us, as we have applied it to more than a hundred operations, yielding more than a thousand routines that are part of the libflame library. We use the solution of the triangular Lyapunov equation as our motivating example. A reader who is not well-versed in linear algebra should not worry: the methodology is systematic to the point where one does not need to be an expert in order to apply it. A more basic treatment that targets novices and has been used at the undergraduate level can be found in [26].

2.1 The FLAME methodology

A fundamental insight in our project was the realization that the Fundamental Invariance Theorem [16], used to prove the correctness of a loop in a program, can be formulated as a worksheet that is systematically filled out, first with assertions (predicates) and then with commands (imperative statements) [3]. The worksheet, given in Figure 1, will be filled out with predicates that indicate prescribed states and commands that achieve those states. It is filled out in the order indicated in the column marked by Step. The predicates $P_{\text{pre}}, P_{\text{post}}, P_{\text{inv}},$ and $G$ represent the precondition, postcondition, loop-invariant, and loop-guard, respectively. The loop-invariant has to be true in four different places: before and after the loop, and at the top and bottom of the loop body. The other parts of the worksheet will become obvious as we fill it out for our example.

Figure 1: Blank FLAME worksheet used to derive algorithms.
2.2 Example: the solution of the triangular Lyapunov equation

We now show how the methodology is applied to a prototypical example: the solution of the triangular Lyapunov equation given by $U^T X + XU + C = 0$ (or, alternatively, $U^T X + XU = -C$), where $U$ is an upper triangular matrix and $C$ and $X$ are symmetric matrices. Here the superscript ‘$T$’ indicates matrix transposition. The solution $X$ is to be computed. Because of symmetry, only the upper triangular part of $C$ is stored and is overwritten with the upper triangular part of $X$. This operation is preceded by a pre-processing phase (not discussed in this paper) that transforms the general (non-triangular) time-continuous Lyapunov equation (an operation encountered in control theory [21]) to the given triangular Lyapunov form.

2.3 Filling out the worksheet

At this point, the reader should imagine the worksheet in Figure 2 as being empty and the steps detailed below as filling out the worksheet in the indicated order.

**Step 1: The precondition and postcondition.**

The precondition is given by $C = \hat{C}$ while the postcondition is $C = X \land U^T X + XU = -\hat{C}$. Here the `\^` is needed to be able to reason about the current contents (state) of variable $C$ relative to the initial contents, $\hat{C}$. For brevity, properties of the matrices (like triangular structure and sizes) are not expressed in the precondition nor in other predicates.

**Step 2: Deriving the loop-invariants.** A fundamental insight is that many algorithms sweep through matrices (arrays) in a systematic and predictable fashion. In this example, all matrices are either triangular or symmetric and are partitioned into quadrants since this exposes regions of the matrices that are either (implicitly or explicitly) zero, for the triangular matrix, or not stored, for the symmetric matrices:

$$ U = \begin{pmatrix} U_{TL} & U_{TR} \\ 0 & U_{BR} \end{pmatrix}, \quad X = \begin{pmatrix} X_{TL} & X_{TR} \\ \ast & X_{BR} \end{pmatrix}, \quad C = \begin{pmatrix} C_{TL} & C_{TR} \\ \ast & C_{BR} \end{pmatrix}, \quad \text{and} \quad \hat{C} = \begin{pmatrix} \hat{C}_{TL} & \hat{C}_{TR} \\ \ast & \hat{C}_{BR} \end{pmatrix}, \quad (1) $$

where $U_{TL}, X_{TL}, C_{TL}$, and $\hat{C}_{TL}$ are all conformal (of the same size) and square. Here $TL, TR, BL$, and $BR$ stand for “top-left”, “top-right”, “bottom-left”, and “bottom-right”, respectively. The $0$ and $\ast$ indicate submatrices that are entirely zero or not stored, respectively. As the algorithm progresses, the top-left (TL) quadrants will grow from empty $(0 \times 0)$ to encompassing the entire matrix.

Substituting the partitioned matrices in Equation 1 into the postcondition yields the expression in Figure 3, which we call the Partitioned Matrix Expression [2, 26] (PME). It is a recursive definition of the operation in terms of the exposed submatrices.

The PME expresses the computation to be performed (which must make the postcondition hold true) in terms of the quadrants. Observe that as long as the loop has not finished, only part of the computation expressed by the PME is satisfied: to come up with potential loop-invariants, one deletes some of the subexpressions from the PME, as illustrated in Figure 4. As long as the expression that is left is a valid expression that has the correct size, it is a candidate. Some potential loop-invariants are such that subsequent steps cannot be performed, which means that they do not yield an (admissible) algorithm. For example, if only the original contents of the matrix are left after deleting subexpressions from the PME, the loop can clearly not complete in a state where the postcondition holds; this exhibits itself when no loop-guard can be found in Step 3. Invariants are thus systematically derived from the PME.

We will now focus on one loop-invariant (Loop-invariant 3) as we fill out the remainder of the worksheet in Figure 2. The methodology yields algorithms corresponding to the other loop-invariants in an analogous fashion.

**Step 3: Loop guard $G$.** We know that after the loop completes, $\{P_{pre} \land \neg G\}$ is true. No commands exists between this and the postcondition $\{P_{post}\}$. Thus, $G$ must be chosen so that

$$ \begin{pmatrix} C_{TL} & C_{TR} \\ \ast & C_{BR} \end{pmatrix} = \begin{pmatrix} X_{TL} & X_{TR} \\ \ast & X_{BR} \end{pmatrix} \land \begin{pmatrix} U_{TL}^T X_{TL} + X_{TL}U_{TL} = -\hat{C}_{TL} \\ U_{TL}^T X_{TR} + X_{TR}U_{BR} = -\hat{C}_{TR} - X_{TL}U_{TR} \\ X_{BR} = \hat{C}_{BR} \end{pmatrix} \land \neg G $$
Figure 2: Worksheet for deriving the unblocked algorithm for solving the triangular Lyapunov equation corresponding to Loop-invariant 3.

implies $U^TX + XU = -C$. This yields the (nonunique) choice for the loop-guard: $G = (m(U_{TL}) < m(U))$, where $m(\cdot)$ returns the row dimension of the argument. (An implicit assumption here is that matrices $U_{TL}$, $X_{TL}$, $C_{TL}$, and $C_{TR}$ are always kept conformal, i.e., of the same size and square.)
Substituting the partitioned operands in (1) into $U^TX + XU = -\mathcal{C}$ (the postcondition), yields

\[
\begin{pmatrix}
C_{TL} & C_{TR} \\
* & C_{BR}
\end{pmatrix}
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
= 
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
\begin{pmatrix}
U_{TL} & U_{TR} \\
0 & U_{BR}
\end{pmatrix}
\]

\[\wedge \begin{pmatrix}
U_{TL}^T & 0 \\
U_{TR}^T & U_{BR}^T
\end{pmatrix}
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
+ 
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
\begin{pmatrix}
0 & U_{TR}^T \\
U_{TL} & U_{BR}^T
\end{pmatrix}
= 
\begin{pmatrix}
-\mathcal{C}_{TL} & -\mathcal{C}_{TR} \\
* & -\mathcal{C}_{BR}
\end{pmatrix}
\]

which (by linear algebra manipulation) is equivalent to

\[
\begin{pmatrix}
C_{TL} & C_{TR} \\
* & C_{BR}
\end{pmatrix}
= 
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
\begin{pmatrix}
U_{TL}^T & X_{TL}U_{TL} \\
U_{TR}^T & X_{TR}U_{TR}
\end{pmatrix}
\]

\[\wedge \begin{pmatrix}
U_{TL}^T & X_{TL}U_{TL} \\
U_{TR}^T & X_{TR}U_{TR} + X_{TR}U_{BR}
\end{pmatrix}
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
+ 
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
\begin{pmatrix}
U_{TL}^T & X_{TL}U_{TR} \\
0 & X_{TR}^T
\end{pmatrix}
= 
\begin{pmatrix}
-\mathcal{C}_{TL} & -\mathcal{C}_{TR} \\
* & -\mathcal{C}_{BR}
\end{pmatrix}
\]

Figure 3: The Partitioned Matrix Expression (PME) (recursive definition of the operation) for overwriting $C$ with the solution of the triangular Lyapunov equation.

---

<table>
<thead>
<tr>
<th>Loop-invariant 1:</th>
<th>Loop-invariant 2:</th>
<th>Loop-invariant 3:</th>
<th>Loop-invariant 4:</th>
</tr>
</thead>
</table>
| \[
\begin{pmatrix}
C_{TL} & C_{TR} \\
* & C_{BR}
\end{pmatrix}
= 
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
\begin{pmatrix}
U_{TL} & U_{TR} \\
0 & U_{BR}
\end{pmatrix}
\]
| \[\wedge \begin{pmatrix}
U_{TL}^T & 0 \\
U_{TR}^T & U_{BR}^T
\end{pmatrix}
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
+ 
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
\begin{pmatrix}
0 & U_{TR}^T \\
U_{TL} & U_{BR}^T
\end{pmatrix}
= 
\begin{pmatrix}
-\mathcal{C}_{TL} & -\mathcal{C}_{TR} \\
* & -\mathcal{C}_{BR}
\end{pmatrix}
\]
| \[
\begin{pmatrix}
C_{TL} & C_{TR} \\
* & C_{BR}
\end{pmatrix}
= 
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
\begin{pmatrix}
U_{TL} & X_{TL}U_{TL} \\
U_{TR} & X_{TR}U_{TR} + X_{TR}U_{BR}
\end{pmatrix}
\]
| \[\wedge \begin{pmatrix}
U_{TL}^T & X_{TL}U_{TL} \\
U_{TR}^T & X_{TR}U_{TR} + X_{TR}U_{BR}
\end{pmatrix}
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
+ 
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
\begin{pmatrix}
U_{TL}^T & X_{TL}U_{TR} \\
0 & X_{TR}^T
\end{pmatrix}
= 
\begin{pmatrix}
-\mathcal{C}_{TL} & -\mathcal{C}_{TR} \\
* & -\mathcal{C}_{BR}
\end{pmatrix}
\]

Figure 4: Loop-invariants for the triangular Lyapunov equation.

**Step 4: Initialization.** The initialization is an indexing step: the matrices are partitioned as in (1). The fact that this must place the variables in a state where $P_{we}$ holds dictates the choice wherein the top-left quadrants are $0 \times 0$ (empty). There could be other choices for the initialization, but those would invariably involve performing computation with the matrices, altering their contents.

**Step 5: Moving through the matrices.** In Steps 5a and 5b, submatrices are exposed so that we make progress through the matrices. Here, thick lines have semantic meaning: a new row and column are exposed. Updates will
happen in the loop body, and then that row and column are moved across the thick line to capture the traversal through the matrices.

In exposing submatrices, we use notational conventions that allow special properties of the submatrices to be easily recognized: lower case Greek letters denote scalars, lower case Roman letters denote column vectors, and upper case Roman letter denote matrices. Submatrices like \( u_{12}^T \) can be easily recognized as being part of a row and hence a row vector (transposed column vector).

The fact that the top-left quadrant is initially empty (0×0) and must eventually envelop the entire matrix dictates how the algorithm traverses through the matrices. The traversal through the matrix, together with the finite size of the operands, means that there is a natural, monotonically decreasing loop-bound function, \( t = n - m(U_{TL}) \), that is bounded below. This ensures that the loop terminates.

**Step 6: State before the update.** The commands in Step 5a are merely indexing operations. Since no computation occurs between the top of the loop and Step 6 in the worksheet, the state of the submatrices that are exposed by Step 5a can be determined by textual substitution and linear algebra manipulation, as illustrated in Figure 5. This yields the state described by Step 6. The invariant together with the partitioning in Step 5a dictates the predicate in Step 7.

**Step 7: State after the update.** Similarly, the commands in Step 5b are merely indexing operations. Since the invariant must again hold, the state in Step 7 can be systematically derived by textual substitution of the submatrices in Step 5b into the loop-invariant and linear algebra manipulation, as illustrated in Figure 6. The loop-invariant together with the redefinition of the quadrants in Step 5b dictate the predicate in Step 7.

**Step 8: Update.** The update in Step 8 is now dictated by the state that the variables are in at Step 6 and the state that they must be in at Step 7:

- \( C_{00} \) already contains \( X_{00} \), the solution to \( U_{00}^T X_{00} + X_{00} U_{00} = -\hat{C}_{00} \), and hence is not updated.
- \( c_{01} \) already contains \( x_{01} \), the solution to \( U_{00}^T x_{01} + X_{00} u_{01} + x_{01} v_{11} = -\hat{c}_{01} \), and hence is not updated.
- \( C_{02} \) already contains \( X_{02} \), the solution to \( U_{00}^T X_{02} + X_{00} U_{02} + x_{01} u_{12}^T + X_{02} U_{22} = -\hat{C}_{02} \), and hence is not updated.
- \( \gamma_{11} \) contains \( \chi_{11} = \hat{\gamma}_{11} \) and needs to be overwritten by the solution, \( \chi_{11} \), of \( 2u_{01}^T x_{01} + 2v_{11} \chi_{11} = -\hat{\gamma}_{11} \). Recognizing that at this point \( c_{01} \) contains \( x_{01} \) and \( \gamma_{11} \) contains \( \hat{\gamma}_{11} \), this can be accomplished by updating \( \gamma_{11} \) with
  \[
  \gamma_{11} := (-\gamma_{11} - 2u_{01}^T c_{01})/(2v_{11}),
  \]
  where \( := \) is used for assignment.
- \( c_{12}^T \) holds \( x_{12}^T = \hat{c}_{12}^T \) and needs to be overwritten with the solution, \( x_{12}^T \), of \( u_{01}^T X_{02} + v_{11} x_{12}^T + x_{01} U_{02} + \chi_{11} u_{12}^T + x_{12}^T U_{22} = -\hat{c}_{12} \). Recognizing that \( X_{02} \) has overwritten \( C_{02} \), etc., this can be accomplished by updating it with the solution, \( x_{12}^T \), of \( v_{11} x_{12}^T + x_{12}^T U_{22} = -\hat{c}_{12} - u_{01}^T C_{02} - u_{01}^T U_{02} - \gamma_{11} u_{12}^T \):
  - First, we update \( \hat{c}_{12} := -\hat{c}_{12} - u_{01}^T C_{02} - u_{01}^T U_{02} - \gamma_{11} u_{12}^T \).
  - Next, we compute \( c_{12}^T := c_{12}^T (v_{11} I + U_{22})^{-1} \). This requires the solution of a triangular system of equations, since it is equivalent to computing the solution to \( (v_{11} I + U_{22})^T x_{12} = c_{12} \) and overwriting \( c_{12}^T \) with \( x_{12}^T \).

We reiterate that the state in Step 6 and the desired state in Step 7 dictate how the variables (submatrices of \( C \)) must be updated.

**Resulting algorithm.** The resulting algorithm, stripped of the annotations that were used to derive it, is given in Figure 7 (left), executing only the commands indicated under Variant 3.

## 2.4 Other algorithms

Other algorithms are derived from the other loop-invariants. In addition, blocked algorithms, which cast most computation in terms of matrix-matrix operations and hence can attain higher performance, can be derived by moving through the matrix several rows and columns at a time. Resulting algorithms are given in Figure 7. In the blocked algorithms, the operation
Step 5a, given by

\[
\begin{pmatrix}
U_{TL} & U_{TR} \\
0 & U_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
U_{00} & u_{01} & U_{02} \\
v_{11} & u_{12}^T & 0 \\
0 & 0 & U_{22}
\end{pmatrix},
\begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
X_{00} & x_{01} & X_{02} \\
* & x_{12}^T & X_{22}
\end{pmatrix},
\begin{pmatrix}
C_{TL} & C_{TR} \\
* & C_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
C_{00} & c_{01} & C_{02} \\
* & \gamma_{11} & C_{12} \\
* & * & C_{22}
\end{pmatrix},
\begin{pmatrix}
\hat{C}_{TL} & \hat{C}_{TR} \\
* & \hat{C}_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
\hat{C}_{00} & \hat{c}_{01} & \hat{C}_{02} \\
* & \hat{\gamma}_{11} & \hat{C}_{12} \\
* & * & \hat{C}_{22}
\end{pmatrix},
\]

expresses

\[
\begin{pmatrix}
U_{TL} & U_{00} \\
0 & U_{TR} \\
\end{pmatrix}
\rightarrow
\begin{pmatrix}
U_{00} & u_{01} & U_{02} \\
v_{11} & u_{12}^T & 0 \\
0 & 0 & U_{22}
\end{pmatrix},
\begin{pmatrix}
X_{TL} & X_{00} \\
X_{TR} & X_{02}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
X_{00} & x_{01} & X_{02} \\
* & x_{12}^T & X_{22}
\end{pmatrix},
\begin{pmatrix}
C_{TL} & C_{00} \\
C_{TR} & C_{02} \\
* & C_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
C_{00} & c_{01} & C_{02} \\
\gamma_{11} & C_{12} \\
* & C_{22}
\end{pmatrix},
\begin{pmatrix}
\hat{C}_{TL} & \hat{C}_{00} \\
\hat{C}_{TR} & \hat{C}_{02} \\
\hat{C}_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
\hat{C}_{00} & \hat{c}_{01} & \hat{C}_{02} \\
\hat{\gamma}_{11} & \hat{C}_{12} \\
\hat{C}_{22}
\end{pmatrix}.
\]

Substituting these into the state of the variables at the top of the loop (the invariant) given by

\[
\begin{pmatrix}
C_{TL} & C_{TR} \\
* & C_{BR}
\end{pmatrix} = \begin{pmatrix}
X_{TL} & X_{TR} \\
* & X_{BR}
\end{pmatrix} \land \begin{cases}
U_{TL}X_{TL} + X_{TL}U_{TL} = -\hat{C}_{TL} \\
U_{TR}X_{TR} + X_{TR}U_{BR} = -\hat{C}_{TR} - X_{TL}U_{TR} \\
X_{BR} = \hat{C}_{BR}
\end{cases}
\]

yields the state of the exposed submatrices:

\[
\begin{pmatrix}
C_{00} & c_{01} & C_{02} \\
* & \gamma_{11} & C_{12} \\
* & * & C_{22}
\end{pmatrix} = \begin{pmatrix}
X_{00} & x_{01} & X_{02} \\
* & \chi_{11} & X_{12} \\
* & * & X_{22}
\end{pmatrix},
\]

Algebraic manipulation of the above expression yields Step 6 in the worksheet.

Figure 5: Systematic derivation of the state of the variables at Step 6 in Figure 2.

\[
Solve \ U_{IJ}^TX_{IJ} + X_{IJ}U_{JJ} = C_{IJ}
\]

requires the solution of the Sylvester equation. Algorithms for that operation were derived in [25].

### 2.5 Discussion

It is the notation we use that facilitates the derivation process: by presenting submatrices rather than index ranges, the derived algorithm avoids much of the indexing clutter (both in the presentation of the algorithm and the details of the proof of correctness) that is typically found in conventional loop-based algorithms. Indeed, the algorithm exposes only one loop even though it requires approximately \( n^3 \) floating point operations. The other loops are hidden inside
of the linear algebra operations that form the body of the loop. Algorithms for these operations themselves can be, and have been, formally derived, using the same techniques as described in this paper.

2.6 From algorithm to code

Using a correct algorithm as the basis for your implementation does not guarantee that the resulting code will be correct. To preserve the correctness of the algorithm as we translate it to code, we defined APIs for different languages such that the code closely resembles the algorithm [5, 26, 27]. An example of this is given in Figure 8. In that figure, unblocked Variant 3 is coded in M-script, the scripting language of Matlab [23].

We created a handful of routines that partition and repartition the matrices. White-space is used to make the code resemble the algorithm as closely as possible. The code in Figure 8 gave the correct answer the first time it was run. We have developed APIs for the C programming language [5, 26, 27, 28]. Additionally, the recently developed Elemental library for distributed memory architectures uses a similar API [24].
Figure 7: Algorithms for computing the solution to the triangular Lyapunov equation. Left: unblocked algorithms. Right: blocked algorithms.

3 Performance

In Figure 9, we show the performance of different algorithmic variants that are implemented as part of libflame, blocked and unblocked, on a Dell PowerEdge R900 server, using four cores and double precision arithmetic. We report performance in GFLOPS (billions of floating point operations per second) by taking the known floating point operation count for this operation, $n^3$, dividing it by the time required to complete the computation, and then scaling.
function [ out ] = lyap_unb_var3( U, C )
    UTL, UTR, ...
    UBL, UBR ] = FLA_Part_2x2( U, ...
        0, 0, 'FLA_TL' );
    CTL, CTR, ...
    CBL, CBR ] = FLA_Part_2x2( C, ...
        0, 0, 'FLA_TL' );
    while ( size( CTL, 1 ) < size( C, 1 ) )
        U00, u01, U02, ...
        u0t, upsilon11, u1t, ...
        U20, u21, U22 ] = ...
            FLA_Repart_2x2_to_3x3( UTL, UTR, ...
                UBL, UBR, ...
                1, 1, 'FLA_BR' );
        CTL, CTR, ...
        CBL, CBR ] = FLA_Cont_with_3x3_to_2x2( ...
            C00, c01, C02, ...
            c0t, gamma11, c1t, ...
            C20, c21, C22 ] = ...
                FLA_Repart_2x2_to_3x3( CTL, CTR, ...
                    CBL, CBR, ...
                    1, 1, 'FLA_BR' );
    end
    C_out = [ CTL, CTR
        CBL, CBR ];
    return

Figure 8: M-script implementation of unblocked Variant 3.

The commands in the body of the loop map are implemented as calls to Basic Linear Algebra Subprograms (BLAS) [22, 10, 9], an interface to commonly encountered linear algebra operations, as well as other routines supported by libflame, which themselves call BLAS operations. As part of our project, we have derived a full library of these operations, but for this experiment we are depending on optimized implementations provided by the GotoBLAS2 implementation [15, 14]. For the blocked algorithms, a block size of 128 was used. While most of our papers focus on achieving high performance, this paper does not and hence we do not go into further details regarding the experiments.

The important thing to note is that distinct variants give rise to different performance and that therefore there is a benefit to identifying all algorithmic variants. For other operations, targeting sequential, multithreaded, and distributed memory architectures, we have consistently shown the benefit of having a choice of algorithms. Several illuminating examples, including derivations and performance comparisons, can be found in [18, 3, 25, 26, 4].

4 Past, Current, and Future Directions

This paper gives a refined presentation of the methodology already proposed in [3, 26]. In this section, we discuss the developments that this work has enabled over the last decade, the current impact of the project, and future directions.
Figure 9: Performance of the various algorithms, on a four core architecture.

Mechanical derivation of linear algebra libraries. Early on in the project it was recognized that with the introduction of the worksheet, the methodology became systematic to the point where it could be automated. This led to a mechanical system, implemented in Mathematica, that does just that [2]. For an operation like the solution of a triangular Lyapunov equation, that system performs the steps described in Section 2 mechanically, provided that a loop-invariant is given by the user. The system outputs an algorithm, M-script implementation, and C implementation. More recently, a system that can automatically generate PMEs has been developed [13], as well as one that then automatically generates loop-invariants [12]. Combining these efforts would yield a system that, given a linear algebra operation, would automate all steps described in this paper.

Scope. As part of the libflame library, the illustrated methodology has been applied to all operations supported by the BLAS and a large number of operations supported by the LAPACK library. In all, the libflame library, written in C, comprises about 1500 implementations of algorithms for about 150 distinct operations (operations that differ in whether they work, for example, with upper or lower triangular matrices are counted separately). Most of these were formally derived as described in this paper.

Formal derivation of stability analyses. The notion of correctness of an algorithm is somewhat different in the presence of round-off error. There, we must instead establish that an algorithm is numerically stable. We have extended the derivation process and worksheet described in this paper so that once algorithms have been derived, numerical stability results can be systematically derived in a similar fashion [2, 6].

Sparse linear solvers. As mentioned in the introduction, sparse iterative solvers for linear algebra problems are more commonly used by applications. These are based on Krylov subspace methods, the Conjugate Gradient method being one example. Recently, we showed that the formal derivation methodology we developed can be extended to the formal derivation of these Krylov subspace methods [11].
5 Conclusion

We have demonstrated how formal derivation of loop-based algorithms is both viable and valuable for the domain of dense linear algebra. Key to the success of the methodology has been a notation that hides indexing details when using arrays and subarrays, the definition of an operation to be implemented via the Partitioned Matrix Expression (PME), a systematic way for identifying loop-invariants, and a framework (the worksheet) that captures the Fundamental Invariance Theorem in a way that clearly links it to a loop-based algorithm.

Additional information

In this document, we cite a large number of our own papers. This is intended to convey how broadly applicable the methodology is and the extent of its practical impact. This, however, obscures what papers are particularly important for a reader who wants to learn more. We recommend first reading the paper in which the worksheet was initially proposed [3] and the book that we wrote for a general audience, at the undergraduate level [26]. The FLAME project website, http://www.cs.utexas.edu/users/flame/ is also a useful resource.

Acknowledgments

We thank Jay Misra for encouraging us to target the formal methods community with this paper and the remaining members of the FLAME team for their contributions over the last ten years.

Researchers at UT-Austin were supported in part by NSF grants CCF-0342369, CCF-0540926, CCF-0702714, CCF-0704217 (REU Supplement), OCI-0850750, CCF-0917096, CCF-0917167, the UT-Austin Institute for Computational Engineering and Sciences (ICES), Intel, Microsoft, National Instruments (through a grant from Dr. James Truchard), and NEC Systems (America), Inc. Researchers at Universidad Jaume I (Spain) were supported in part by the Spanish Ministry of Science and Innovation through grant TIN2008-06570-C04 and a grant from NVIDIA. Researchers at RWTH Aachen University gratefully acknowledge financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

The example used to illustrate the techniques in this paper, the solution of the triangular Lyapunov equation, was first pursued as a Freshman Research Initiative project that also involved Eileen Martin, Burns Healey, and Nick Wiz.

References


