The Dissertation Committee for Paolo Bientinesi
certifies that this is the approved version of the following dissertation:

Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms

Committee:

Robert van de Geijn, Supervisor

Todd Arbogast

Alan Cline

Inderjit Dhillon

Calvin Lin

Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms

by

Paolo Bientinesi, M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2006

A Mamma e Babbo.

In memory of Prof. Milvio Capovani,
to whom I will always be indebted,
for making me fall in love with matrix computations.

Acknowledgments

When discussing professors' advising styles, I point out that there is no one best style. It is the relationship between advisor and student that matters. In this respect, the relationship that I have shared with Robert in the past six years has been superlative. Together, we have identified and discussed dozens and dozens of problems. Often we have also found solutions to them. Not once have I felt that I had to adhere to Robert's opinions because of the professor-student hierarchy. Quite the opposite instead: Robert has always respected me and my ideas as if we were colleagues. He gave me the opportunity to pursue my own goals while steering me towards interesting and worthwhile objectives. In any situation, he has always acted with the intent of doing what was best for my career. With him on my side, I entered the PhD program learning about high-performance computing, and I have come out being ready for an academic career. In short, Robert for me has been neither a father figure or a buddy: he has been the best advisor I could possibly have asked for.

A big thank you goes to Marco Pellegrini. If there is one person responsible for my embarking on the whole PhD adventure, it is Marco. Early in 1998 I completed my *laurea* thesis under his supervision. Immediately thereafter, not only did he strongly recommend and support my application to American PhD programs, but he also made it possible for me to spend an early semester in the US. That way I could get a taste of the different school system and culture. Priceless. Another big

thank you goes to everybody at the Institute of Computational Mathematics (IMC) in Pisa, who have always welcomed and hosted me as if I were part of the family.

Writing a coherent document that summarizes results obtained over five years[1] is not necessarily a trivial task. Indeed, for me it was far from being trivial. I want to thank my dissertation committee and John Gunnels for providing me with suggestions and comments. I reserve a solo thank you for Prof. Todd Arbogast for taking the time to meticulously inspect this dissertation, pointing out the many imperfections.

I have to acknowledge the regulars of the FLAME research group: Enrique Quintana-Orti, Brian Gunter, Kazushige Goto, Tze Meng Low, Field van Zee, Ernie Chan, Kent Milfeld, Jim Nagle, Avi Purkayastha and Victor Eijkhout. Many times I used them as guinea pigs for my talks and lectures. Many other times I learned from them about the latest and greatest in high-performance computing. Thank you guys. Our weekly research meetings have been an extraordinarily effective environment for sharing, learning and teaching ideas.

The past six years in Austin have been increasingly pleasant. This is probably due to an ever growing list of friends. It would be impossible to mention them all here. Strictly in alphabetical order, I want to name the ones that have accompanied me throughout the entire experience: Brownie (Brian Dias), Goofy (Jeff Napper), Focker (Suvrit Sra) and Shy-mon (Shimon Whiteson). With them I spent 1) countless hours playing (and winning) many games and sports, 2) countless words chatting, discussing science and anything else, 3) countless dollars going out to eat. I wish you guys will be writing these pages sometime within the next two years. I also want to send a Ciao to everybody in Sweetlife: it is now more than six

[1]This research has been funded by the following NSF grants. Award CCF-0342369: Automatic Tools for Deriving, Analyzing, and Implementing Linear Algebra Libraries; Award ACI-0305163: Collaborative Research: A Systematic Approach to the Derivation, Representation, Analysis, and Correctness of Dense and Banded Linear Algebra Algorithms for HPC Architectures; Award ACI-0203685: Collaborative Research: New Contributions to the Theory and Practice of Programming Linear Algebra Libraries.

years that we regularly meet every Thursday evening! Isn't it amazing?

A separate note for my friends in Italy. Among them, are friends from my hometown, from my school, from my university and from the Navy. These are guys I do not need to explicitly name, they know who they are. Distance and time do not matter for them. When we meet it is as if we had been together until the day before, as if I had never left for the US. I don't have a specific reason for thanking you guys, if not for being the ones on whom I will always be able to count. I wish teleport really existed.

Infine un ringraziamento in italiano per i miei genitori. Mamma e Babbo hanno sempre fatto di tutto per mettermi nelle condizioni ottimali per dedicarmi solamente ai miei studi. Mi hanno sempre aiutato ed appoggiato, seppure la mia decisione di lasciare l'Italia per loro fosse tutto fuorche' una buona notizia. Grazie. A ruota segue un ringraziamento a mia sorella Aba che si e' sempre presa cura dei miei genitori durante questa mia prolungata assenza.
(Finally, an Italian thank you to my parents. Mom and Dad always created the best environment for me to only focus on studying. They always supported and encouraged me, even though my decision to leave Italy for them was anything but good news. Thank you. I dedicate this dissertation to both of you. This leads me to also thank my sister Aba, for taking care of my parents during my extended leave.)

Paolo Bientinesi

The University of Texas at Austin
August 2006

Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms

Publication No. _____

Paolo Bientinesi, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Robert van de Geijn

We consider the problem of developing formally correct dense linear algebra libraries. The problem would be solved convincingly if, starting from the mathematical specification of a target operation, it were possible to generate, implement and analyze a family of correct algorithms that compute the operation. This thesis presents evidence that for a class of dense linear operations, systematic and mechanical development of algorithms is within reach. It describes and demonstrates an approach for deriving and implementing, systematically and even mechanically, proven correct algorithms. It also introduces a systematic procedure to analyze, in a modular fashion, numerical properties of the generated algorithms.

Contents

Chapter 1

Introduction

"Programming is one of the most difficult branches of applied mathematics;"
E.W. Dijkstra [EWD 498]

Developing and implementing algorithms is a laborious and error-prone process when performed by a human. This dissertation is motivated by the desire of making the development and the implementation of dense linear algebra algorithms as systematic as possible.

1.1 Goals and Motivations

Proving the correctness of programs is a fundamental problem in the field of computer sciences. Throughout the years, in a quest towards a formalism for proving the correctness of algorithms, a number of scientists have received the ACM Turing Award for their contributions to structured programming, formal methods, and analysis of programs: E.W. Dijkstra (1972), J. Backus (1977), R.W. Floyd (1978), C.A.R. Hoare (1980) and J. Hopcroft (1986). Thanks to the seminal work of these pioneers, the discipline of programming today lies on solid grounds: starting from an algorithm (or routine) $\mathcal{A}$, computer scientists possess the tools to assert whether

$\mathcal{A}$ computes a target function or not (i.e., to prove the formal correctness of $\mathcal{A}$). Yet, the field is not closed. If the algorithm is loop-based, the methodology suffers from a limitation: it relies on the programmer's ability to identify a loop-invariant, a predicate that remains *true* throughout the execution of the loop statement. The proof of correctness of a loop-based algorithm depends on this predicate, but no systematic method for identifying loop-invariants is known. As a consequence of this limitation, a formal proof of correctness is often sought for critical applications only, with loop-invariants manually identified by a human.

In the field of scientific computing, the situation is exacerbated by the fact that most algorithms inherently deal with approximate quantities (real numbers) instead of discrete values. In such a scenario, the concept of formal correctness often translates to those of confidence and stability. The level of confidence of a program is *raised* by explicit testing, that is, by executing the program on a vast number of different inputs, monitoring its termination and accuracy (in some form). In addition, the stability of an algorithm is assessed through a careful analysis, by hand, of the propagation of errors due to approximate arithmetic. These premises justify the tradition of creating new numerical libraries by expanding or modifying existing ones, with the intent of retaining, as much as possible, the level of confidence and the stability analyses that were established for the old libraries.

In this dissertation we examine the problem of developing dense linear algebra libraries consisting of formally correct and numerically stable algorithms. Such algorithms are the building blocks for countless scientific applications, including sparse matrix computations. Because of the impact of dense linear algebra on computational science, a library is subject to a number requirements. Here we present a list of the most important requirements.

- **High-performance.** Performance plays a crucial role in the computational sciences, where faster routines result in the ability of solving larger problems

or computing more accurate solutions.

- **Multiple algorithmic variants.** In Figs. 1.1 and 1.2, we report the sequential and parallel performance of three different algorithmic variants for computing the Cholesky factorization. Additionally, in gray, we report the performance for LAPACK, currently the de facto standard linear algebra library in the scientific community. The parallel performance was measured on a shared memory architecture (a system where two or more identical processors are connected to a single shared main memory, also known as an "SMP system").[1] Variant 1 attains the best sequential performance, but it is the slowest in the parallel environment. By contrast, Variant 3 attains the best parallel performance, and the worst sequential. LAPACK only includes one variant,[2] Variant 2, that is the second best in both scenarios. Fig. 1.3 reports the parallel performance of four variants for computing the inverse of a triangular matrix: the variant included in LAPACK (in grey) is again suboptimal on both sequential and parallel systems; in this example the difference between the best variant and the second best (LAPACK) is significant. Fig. 1.3 also shows that the best performance is attained by different variants depending on the problem size: Variant 2 is the fastest for small and medium-sized matrices, while Variant 3 is to be preferred for very large matrices.

  We conclude that in order to obtain high-performance in different scenarios, a library has to include a family of algorithmic variants computing the same operation.

[1] The experiments were performed on an IBM Power 4 SMP System. This architecture consists of an SMP node, with sixteen 1.3 GHz Power4 processors and 32 GBytes of shared memory. The processors operate at 4 FLOPS per cycle, for a peak theoretical performance of 5.2 GFLOPS/proc (billions of FLOPS, per processor), with a DGEMM (matrix-matrix multiply) benchmarked at 3.7 GFLOPS/proc. We measured performance running the experiments with one and sixteen processors.

[2] LAPACK normally contains two routines per operation: the blocked and unblocked versions.

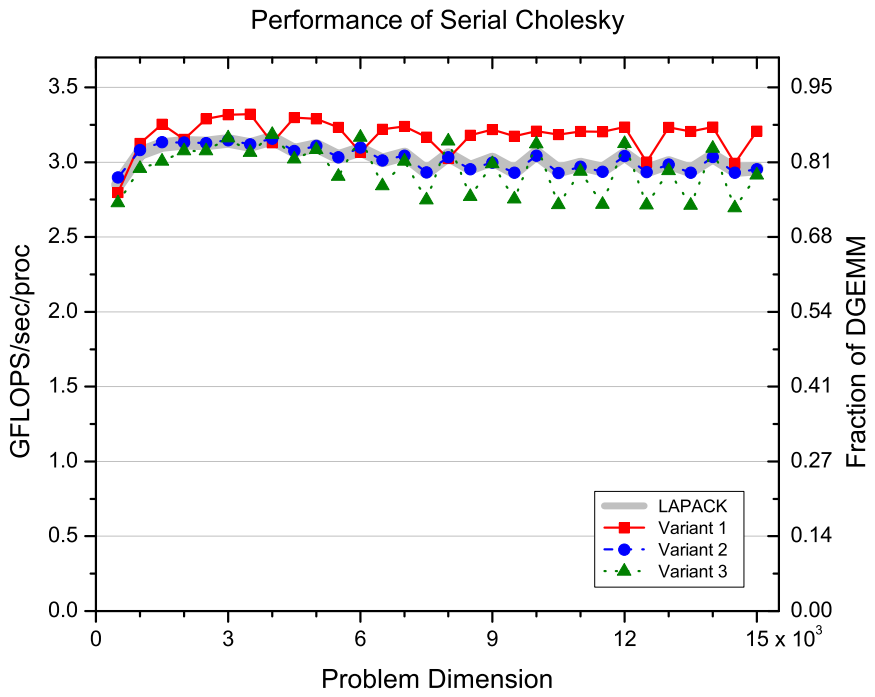

Figure 1.1: Performance for the sequential computation of the Cholesky factorization.

- **Coverage.** A library should include the functionality encountered by scientific applications.

- **Correctness.**

  > "Program testing can be used to show the presence of bugs, but never to show their absence! Therefore, proof of program correctness should depend only upon the program text."
  >
  > E.W. Dijkstra [EWD268]

  When developing numerical routines, it is customary to consider testing as the principal tool for asserting formal correctness. The routines are executed thousands (or even millions) of times on carefully constructed test-beds comprised

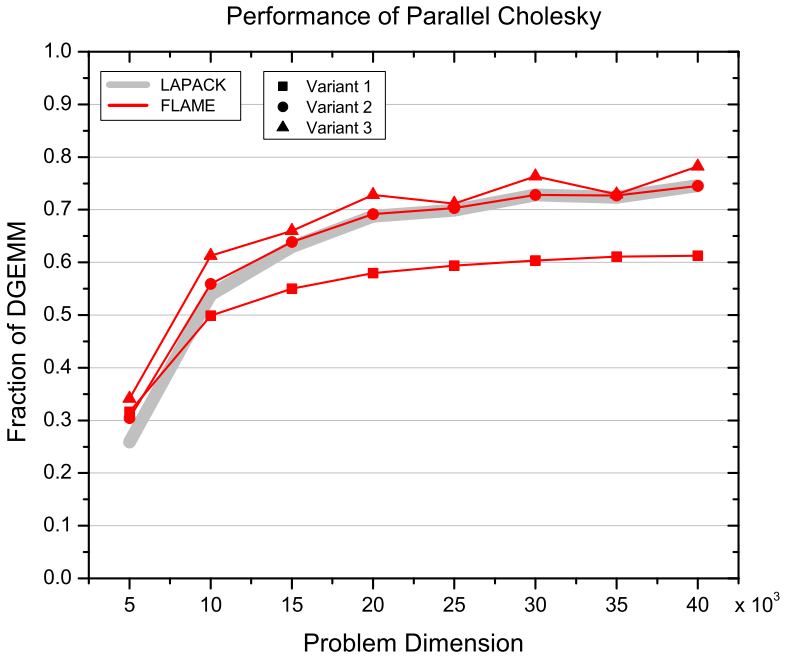

Figure 1.2: Computation of the Cholesky factorization in a shared memory environment: performance for 16 processors.

of artificial matrices with specific properties along with matrices arising in real-life applications. While testing is certainly indispensable, it is not enough to certify the correctness of a routine. A formal proof of correctness, together with testing, would increase dramatically the confidence level of routines and libraries.

- **Stability Analysis.** A numerical algorithm is to be trusted only if documented with a stability analysis: unstable algorithms may return highly inaccurate answers and must not be taken into consideration for performance reasons. Moreover, given the modular structure of scientific applications, the analysis of an algorithm is useful for assessing the stability of the operations that use such an algorithm as a component.

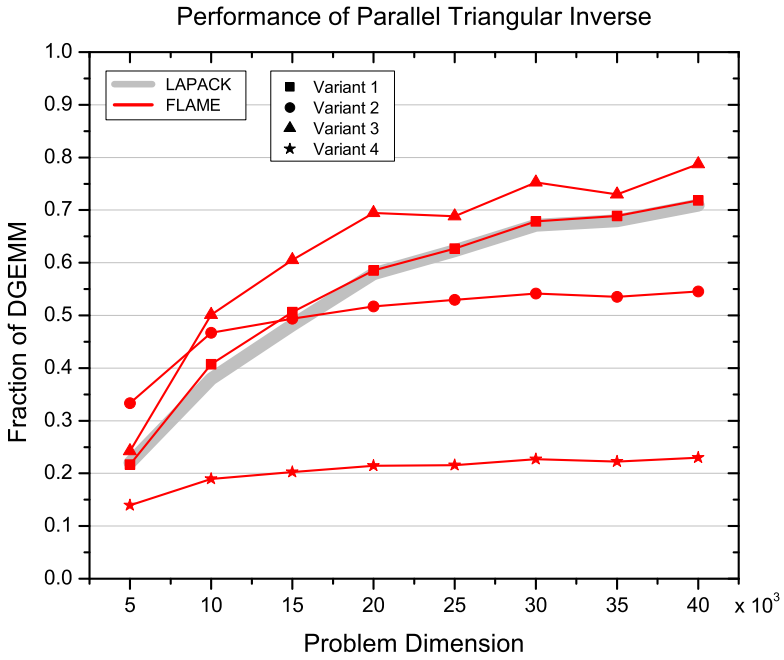

Figure 1.3: Computation of the inverse of a triangular matrix in a shared memory environment: performance for 16 processors.

- **Performance Analysis.** When facing many algorithmic variants for the same operation, the users need criteria to select the best variant for a specific scenario. Every routine in the library should be documented with a performance analysis. Better yet, the library should automatically chose the best algorithm.

Because of the scientific advancements and ever evolving architectures, new libraries, containing new and improved algorithms (customized for different architectures), are always in demand. The development of a library that satisfies the listed requirements represents a daunting task.

Traditionally the problem has been made manageable by reducing the requirements and by adopting an incremental approach (new libraries are built upon

previous ones). Presently, it is common for libraries to include at most one or two algorithmic variants for most operations. Such a simplification significantly affects the task of testing and analyzing. No performance analysis is needed. The stability analysis is often inherited from previous libraries. Furthermore, formal correctness has never been pursued.

We approach the development of dense linear algebra libraries from a different perspective. The magnitude of effort required to develop and maintain a library in accordance with the listed requirements attests to the need for mechanical tools to aid, or even replace, the human programmer. Our vision is to shift the burden from a human to a mechanical system. Such a system would take the mathematical description of a target operation as input and, at the press of a button, would generate a family of formally correct algorithms that compute the operation. Each algorithm would be in the form of a high-level description or a routine implemented in a target language. In addition, each algorithm would come with an accompanying stability and performance analysis.

The goal of this thesis is to create theories and tools for facilitating the mechanical generation of dense linear algebra libraries.

1.2 Background

Several topics in computer sciences come together in this dissertation: high performance computing, formal derivation methods, numerical linear algebra and automatic generation of programs. In the next sections we provide a brief historical overview of these disciplines. We introduce the concept of "formal correctness," distinguishing between correctness and accuracy of algorithms.

1.2.1 Dense Linear Algebra Libraries

The first libraries for linear algebra appeared in the early 1970s; an exhaustive list of libraries released since would be daunting. Here we only mention those projects that became popular and are related to the dissertation.

The "NAG Fortran Library" was the first (commercial) library to be dispatched, in September 1971; it was written in Fortran (and was also available in Algol 60) and included numerical algorithms for a variety of problems ranging well beyond the realm of linear algebra. The next year (1972) EISPACK [35] was released; it was distributed free of charge and only included routines related to the solution of eigenproblems. The developers of EISPACK made an effort to consistently document every routine with a publication presenting the algorithm and discussing its stability analysis.

A first set of Basic Linear Algebra Subprograms (Level 1 BLAS, or BLAS1 [25, 33]) was identified in 1973; the BLAS1 library consisted of vector operations that were optimized for the then popular vector architectures. The rationale was to express more complex linear algebra algorithms in terms of the BLAS1 operations, which were then optimized for different platforms, to attain portable high-performance. This was the philosophy underlying LINPACK [15] (announced in 1974 and released in 1978), the first library for the solution of dense linear systems that gained widespread acceptance. LINPACK was written in Fortran77 and included a variety of algorithms for general matrices as well as for matrices with structure (symmetric, banded, triangular). Today, almost 30 years after its release, the LINPACK legacy is still very tangible: not only are the fastest supercomputers ranked based on the LINPACK benchmark [16] (computation of the solution of a linear system via LU factorization with pivoting), but many libraries are still developed complying with the modular style introduced by the authors of LINPACK.

In the late 1980s the idea of BLAS was extended to include matrix-vector

operations (Level 2 BLAS, or BLAS2 [18], 1988) and eventually matrix-matrix operations (Level 3 BLAS, or BLAS3 [17], 1990). The objective was to attain high-performance on architectures with a hierarchical memory system by amortizing the costly data movement over a large number of floating point operations. At the same time (1987) a new project started, LAPACK [1]: the goal was to rewrite the EISPACK and LINPACK libraries exploiting the Level 3 BLAS, to attain high performance "on shared-memory vector and parallel processors. On these machines, LINPACK and EISPACK are inefficient because their memory access patterns disregard the multi-layered memory hierarchies of the machines, thereby spending too much time moving data instead of doing useful floating-point operations" [32]. LAPACK was written in Fortran77 and since its inception in 1992 has become the most widely used library for high-performance dense linear algebra.

In the 1990s several libraries targeting distributed memory architectures appeared. In 1993, LAPACK's sister library, ScaLAPACK [12], was released, but because of the cumbersome interface, it never equaled the popularity of LAPACK; it was still written in Fortran77. PLAPACK [36] was released in 1997: this package was not meant as a complete dense linear algebra library per se, but as a convenient infrastructure for coding dense linear algebra algorithms on distributed memory architectures. It was implemented in C and provided the user with a high-level interface, hiding the intricate indexing present in ScaLAPACK. The FLAME Application Program Interfaces (APIs) illustrated in Section 1.3 evolved from PLAPACK.

Starting from 1995 the concept of automatic generation of programs gained popularity. Recognizing that the matrix-matrix multiply (GEMM) is the central operation in dense linear algebra (every other BLAS3 routine is written in terms of it), the PHiPAC project [11], and later the ATLAS [37, 38] project, tackled the problem of optimizing this routine for a target architecture without any machine-

specific hand tuning.[3] The PHiPAC methodology consists of three components: 1) a model for the C compiler that provides guidelines to produce portable high-performance ANSI C code; 2) a parameterized code generator that produces codes according to guidelines; 3) a set of scripts that automatically tune code for a particular architecture by varying the code generators parameters and benchmarking the resulting routines. ATLAS employed a modification of PHiPAC's technique in that the search space is reduced by constraining the number of different implementations that are generated in the search process. Consequently the optimization process completes more quickly, typically in a matter of hours. On the other hand, in terms of performance, these two projects have not delivered routines that approach the performance of hand-optimized BLAS. Currently, the fastest Open Source BLAS implementation for a number of architectures is the GotoBLAS [21] library.

We make a comment to differentiate the concepts expressed by the words "automatic" and "mechanical." We associate "automatic" with a system that implements an optimization process over a parametric space. With respect of the generation of algorithms, both PHiPAC and ATLAS automatically produce many different implementations of the same algorithm (GEMM). We refer to the process of searching for the implementation whose execution time is minimal (among the ones generated) as "tuning" for a target algorithm. In contrast, we use the word "mechanical" to indicate a system that generates different algorithmic variants with no human intervention (each variant can then be optimized by tuning).

Finally we mention Matlab [26]. Rather than being a proper library, it is an effective computing environment, especially when it comes to linear algebra operations. In its latest versions, Matlab acts as a user-friendly interface to linear algebra libraries: the user invokes matrix functions through a high-level interface,[4]

[3] Recent versions of ATLAS produce a mix of automatically tuned and user-contributed code.

[4] As an example, the expressions `x=A\b` and `M=lu(A)` correspond to solving the linear system $Ax = b$ and to computing the LU factorization of matrix A, respectively.

which are then translated into internal functions or into calls to external linear algebra libraries like LAPACK, SLICOT [3] and RECSY [29, 30] (these last two are specialized libraries for control theory equations).

1.2.2 Error Analysis of Algorithms

When dealing with numerical algorithms, it is crucial to study the propagation of errors due to floating point representation and arithmetic. Performance issues may be taken into account only when an algorithm is certified to yield an approximation to the solution that is as good as the conditioning of the problem warrants.

James Wilkinson was the first to publish a thorough analysis of the errors arising in a cross section of numerical algorithms. His books *Rounding Errors in Algebraic Processes* [39] and *The Algebraic Eigenvalue Problem* [40], published about 40 years ago, initiated a new discipline in numerical analysis. The problems and the solutions Wilkinson presented were so crucial to the computer science community that he received both the ACM Turing award and the SIAM von Neumann award.

Since Wilkinson's ground-breaking work many books and papers have extensively treated the argument of analyzing the stability properties of algorithms. The majority of numerical linear algebra publications have a section dedicated to error analysis. A comprehensive book on the subject for many operations in linear algebra is Nick Higham's book *Accuracy and Stability of Numerical Algorithms*[5] [27].

Today the area is the domain of a few experts. Normally the error analysis for a new algorithm is a reduction to a standard result or it consists of a composition of known results. Generally speaking, the techniques used to derive error analyses of algorithms present two deficiencies: they are not sufficiently modular and their clarity greatly depends on notation. Quoting Higham [27]:

> "Two reasons why rounding error analysis can be hard to understand

[5] This book covers algorithms in numerical linear algebra with the exception of the area of eigenvalue and singular value computations.

are that, first, there is no standard notation and, second, error analyses are often cluttered with re-derivations of standard results."

1.2.3 Formal Correctness vs. Numerical Stability

Let $\mathcal{O}p : X \to Y$ be a mathematical function and $\mathcal{A}$ an algorithm implementing $\mathcal{O}p$. The output $\check{y}$ is obtained by executing algorithm $\mathcal{A}$ on input $x \in X$: $\check{y} = \mathcal{A}(x)$. We investigate the relation between the computed solution $\check{y}$ and the exact solution $y = \mathcal{O}p(x)$. Naturally it would be desirable that $\check{y}$ equal y for any input x, but since the computer arithmetic for real numbers is approximate, numerical algorithms cannot generally satisfy such a property. The definition of formal correctness is therefore given under the assumption of exact arithmetic (absence of roundoff error); this leads to the possibility of having formally correct algorithms that compute inexact results. For this reason, when dealing with numerical algorithms, two distinct concepts are needed: formal correctness and numerical stability.

The Floyd-Hoare logic [28, 19] is a formal system for reasoning about the correctness of computer programs with the rigor of mathematical logic. The central tool of this logic is the Hoare triple. A triple describes how the execution of a section of code changes the state of the computation. A Hoare triple is of the form

$$\{P\}\ C\ \{Q\}$$

where P and Q are predicates (formulae in predicate logic) and C is a command. P is called the precondition and Q the postcondition. Such a triple is read as: whenever the command C is executed in a state in which P holds, it will terminate and Q holds upon completion.

Definition 1 (Formal Correctness) *Algorithm $\mathcal{A}$ is* **correct** *if in the absence of roundoff it exactly computes $\mathcal{O}p$. That is, for each $x \in X$ it returns $\check{y} = \mathcal{A}(x)$ such that $\check{y} = \mathcal{O}p(x)$.*

Definition 1 can be stated in term of a Hoare's triple as:

$$\{x = \hat{x}; y = \hat{y}\}\ \mathcal{A}\ \{y = \mathcal{O}p(\hat{x})\}$$

which means that if the algorithm $\mathcal{A}$ is executed in a state in which the variable x equals some value $\hat{x}$, then the execution terminates and it leads to a state in which the value of variable y equals the value $\mathcal{O}p(\hat{x})$.

If algorithm $\mathcal{A}$ does not comprise approximate computations, a proof of (formal) correctness guarantees that $\mathcal{A}$ always returns the same output as $\mathcal{O}p$. The reasons why the same conclusion cannot be drawn for numerical algorithms are: 1) not every real number can be stored exactly in a computer and 2) real number computations (floating point computations) incur approximations. In the presence of floating point computations, a formally correct algorithm may return a value $\check{y}$ which may not even be "close" to the exact solution y.

The study of the numerical stability of algorithms finds bounds to express the quality of computed quantities. It is often the case that it is not the distance (absolute or relative) between the computed solution $\check{y} = \mathcal{A}(x)$ and the exact solution $y = \mathcal{O}p(x)$ that sets apart stable (reliable) algorithms from unstable (unreliable) algorithms: a characterizing criterion is given by the backward stability.

Definition 2 (Numerical Stability) *Let $\mathcal{A}$ be an algorithm that computes $\mathcal{O}p$, and let us assume that it possible to bound* a priori *the number of steps necessary for $\mathcal{A}$ to complete. The algorithm $\mathcal{A}$ is said to be* **backward stable** *if for each $x \in X$, in the presence of roundoff, it returns an output $\check{y}$, which is the exact solution of a problem $\mathcal{O}p(\tilde{x})$, where $\tilde{x}$ is a "small" perturbation of the input data x.*

The backward stability of an algorithm $\mathcal{A}$ implementing the operation $\mathcal{O}p$ can be defined in terms of norms as follow: $\mathcal{A}$ is stable if $\forall x \in X$, there exists a "small" quantity ϵ such that $\mathcal{A}(x) = \mathcal{O}p(\tilde{x})$ and $\|x - \tilde{x}\| \leq \epsilon$. The definition of

"small" is context-dependant.

1.3 The Framework: FLAME

The Formal Linear Algebra Methods Environment (FLAME) effort is a project that aims at generating and implementing high-performance linear algebra algorithms. The project encompasses a large number of theoretical and practical tools, some of which are presented in this thesis. At the core is a new notation for expressing dense linear algebra algorithms [10]. This notation has a number of attractive features: (1) it avoids the intricate indexing into the arrays that store the matrices that often obscures the algorithm; (2) it raises the level of abstraction at which the algorithm is represented; (3) it allows different algorithms for the same operation and similar algorithms for different operations to be easily compared and contrasted; and (4) it allows the state (loop-invariant) of the matrix being updated to be concisely expressed.

FLAME APIs for representing algorithms in code have been defined for a number of programming languages [8]. These APIs allow the code to closely resemble the formally correct algorithms so that (1) the implementation requires little effort and (2) chances of introducing coding error are minimized.

As this dissertation demonstrates, the notation supports a step-by-step process for deriving formally correct families of loop-based algorithms [5]. The methodology is sufficiently systematic that it can be made mechanical [7] and it can be extended to numerical stability analysis [9]. The project is also working towards making performance analysis similarly systematic and mechanical [23].

Here we give an overview of the notation and the APIs employed in the FLAME project. A blocked algorithm for computing the Cholesky factorization of a matrix A is used as an example. The complete derivation of this algorithm and a more detailed presentation of the FLAME notation and APIs are provided in

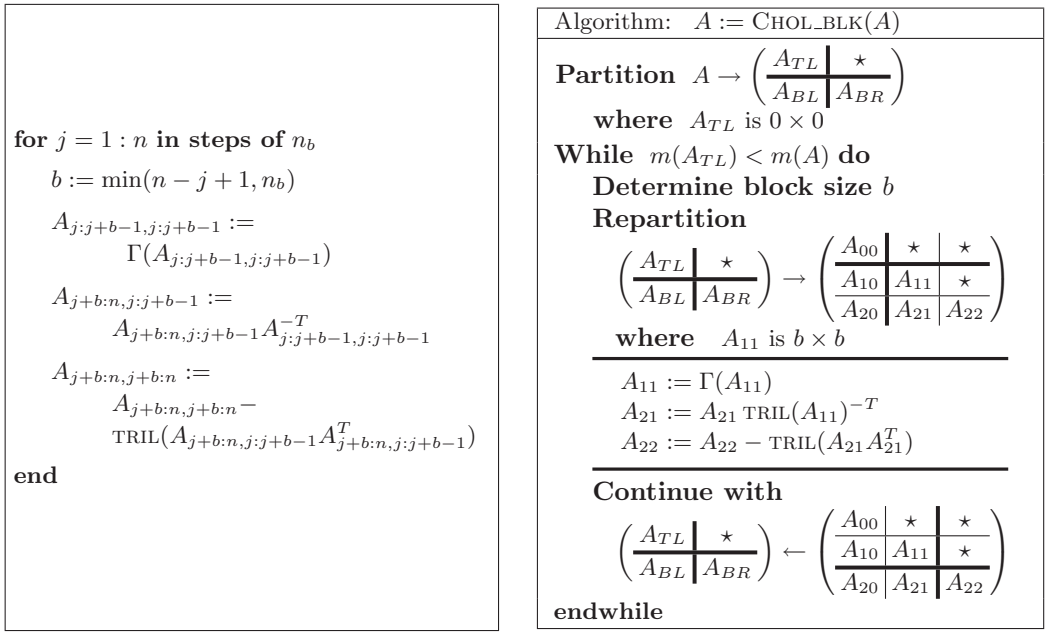

Figure 1.4: Matlab-like and FLAME description of a blocked algorithm for computing the Cholesky factorization. The function $\text{TRIL}(X)$ denotes the lower triangular part of matrix X.

Appendix A.

Fig. 1.4 shows side-by-side, a "Matlab-like" and the FLAME description of the algorithm. It is apparent that the Matlab notation (on the left), although very concise, makes it is hard to identify what regions of the matrix are updated and used. On the contrary, the FLAME notation (on the right) makes these regions explicit. Thanks to a Matlab API ("FLAME@lab"), the FLAME implementation in Fig. 1.5 closely mirrors the algorithm description of Fig. 1.4 (right).

A LAPACK routine implementing the same blocked algorithm is shown in Fig. 1.6. High-performance is attained thanks to the calls to BLAS functions. Again, the code makes it difficult to create a picture of what parts of the matrix are used or updated. A high-performance FLAME implementation (FLAME/C) is given in Fig. 1.7. The routine presents the same structure as the FLAME algorithm

```
function [ A_out ] = Chol_blk( A, nb_alg )
  [ ATL, ATR, ...
    ABL, ABR ] = FLA_Part_2x2( A, ...
                               0, 0, 'FLA_TL' );

  while ( size( ATL, 1 ) < size( A, 1 ) )
    b = min( size( ABR, 1 ), nb_alg );

    [ A00, A01, A02, ...
      A10, A11, A12, ...
      A20, A21, A22 ] = FLA_Repart_2x2_to_3x3( ATL, ATR, ...
                                               ABL, ABR, ...
                                               b, b, 'FLA_BR' );
    %-------------------------------------------------------------%
    A11 = Chol_unb( A11 );
    A21 = A21 / tril( A11 )';
    A22 = A22 - tril( A21 * A21' );
    %-------------------------------------------------------------%
    [ ATL, ATR, ...
      ABL, ABR ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02, ...
                                             A10, A11, A12, ...
                                             A20, A21, A22, ...
                                             'FLA_TL' );
  end
  A_out = [ ATL, ATR
            ABL, ABR ];
return
```

Figure 1.5: FLAME@lab (Matlab) code for a blocked Cholesky factorization.

description (Fig. 1.4) and the Matlab implementation (Fig. 1.5).

```
SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO )

*        Parameters declaration.
[..]
*
      DO 20 J = 1, N, NB
*
*        Update and factorize the current diagonal block and test
*        for non-positive-definiteness.
*
         JB = MIN( NB, N-J+1 )
         CALL DSYRK( 'Lower', 'No transpose', JB, J-1, -ONE,
   $                 A( J, 1 ), LDA, ONE, A( J, J ), LDA )
         CALL DPOTF2( 'Lower', JB, A( J, J ), LDA, INFO )
         IF( INFO.NE.0 )
   $        GO TO 30
         IF( J+JB.LE.N ) THEN
*
*           Compute the current block column.
*
            CALL DGEMM( 'No transpose', 'Transpose', N-J-JB+1, JB,
   $                    J-1, -ONE, A( J+JB, 1 ), LDA, A( J, 1 ),
   $                    LDA, ONE, A( J+JB, J ), LDA )
            CALL DTRSM( 'Right', 'Lower', 'Transpose', 'Non-unit',
   $                    N-J-JB+1, JB, ONE, A( J, J ), LDA,
   $                    A( J+JB, J ), LDA )
         END IF
 20   CONTINUE
```

Figure 1.6: LAPACK (Fortran) code for a blocked Cholesky factorization.

```
int FLA_Chol_blk( FLA_Obj A, int nb_alg )
{ /* parameters declaration */
  [..]
  FLA_Part_2x2( A,  &ATL,  &ATR,
                    &ABL,  &ABR,
                      /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );

  while ( FLA_Obj_length( ABR ) != 0 ){
    b = min( FLA_Obj_length( ABR ), nb_alg );

    FLA_Repart_2x2_to_3x3( ATL,  ATR,    &A00, /**/ &A01, &A02,
                                       /* ******************** */
                                         &A10, /**/ &A11, &A12,
                           ABL,  ABR,    &A20, /**/ &A21, &A22,
                 /* with */ b, /* by */ b, /* A11 split from */ FLA_BR );
    /* ************************************************************* */
    FLA_Chol_unb( A11 );

    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE,
              FLA_NONUNIT_DIAG, ONE, A11, A21 );

    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, MINUS_ONE,
              A21, ONE, A22 );
    /* ************************************************************* */
    FLA_Cont_with_3x3_to_2x2( &ATL, &ATR,    A00, A01,/**/ A02,
                                             A10, A11,/**/ A12,
                                           /* *************** */
                              &ABL, &ABR,    A20, A21,/**/ A22,
                              /* with A11 added to submatrix */ FLA_TL );
  }
  return;
}
```

Figure 1.7: FLAME/C code for a blocked Cholesky factorization.

1.4 Contributions

The main contribution of this dissertation is that it provides evidence that a mechanical system for developing linear algebra algorithms and libraries is within reach. By letting a formal correctness proof guide the construction of loop-based algorithms, a systematic methodology is defined. Evidence is presented that this methodology can be made mechanical and that the approach also applies to stability analysis.

Specific contributions to the fields of computer sciences are:

- **Program correctness.** We present a methodology for deriving loop-based algorithms for a large number of dense linear algebra operations. The algorithms are built around a proof of correctness, therefore they are correct by construction and no further proof is required. This result was facilitated by the choice of an appropriate level of abstraction at which to reason about dense linear algebra algorithms. This, in turn, was instrumental for the discovery that for a class of operations, loop-invariants can be identified systematically and *a priori*. Once a loop-invariant is selected, the proof of correctness guides the construction of the corresponding algorithm.

- **Systematic derivation of families of algorithms**. We show that the derivation methodology is systematic and generates many algorithms for the same operation. The methodology consists of a procedure that takes the mathematical specification of a target operation as input and returns a family of algorithms that compute the operation. Every step of the procedure is fully determined by the input.

- **Mechanical derivation.** We identify the information that a mechanical system needs in order to execute the derivation procedure. A prototype system that executes the procedure with limited human intervention is introduced. This system further certifies that the derivation procedure is systematic and

demonstrates that the mechanical derivation of dense linear algebra operations is within reach.

- **Systematic stability analysis.** We present an extension to the derivation procedure for investigating numerical properties of algorithms generated via our methodology. The result is a systematic procedure that allows for modular error analysis.

The contributions to the field of computational science and engineering lie with the **families of algorithms** and the **libraries** that our derivation methodology and tools enable.

1.5 Outline of the Thesis

This dissertation is organized as follows.

Chapter 2 introduces a systematic procedure to derive formally correct loop-based algorithms. A template of a proof of correctness for dense linear algebra algorithms is derived in Sec. 2.1. Such a template becomes a worksheet (Sec. 2.2) for the derivation of algorithms. The derivation is systematic: in Sec. 2.3 an eight-step procedure is described. How to derive multiple algorithms for the same operation is explained in Section 2.5. The derivation methodology discussed in Chapter 2 represents the foundations for both Chapters 3 and 4.

The mechanical generation of algorithms is covered in Chapter 3. In Sec. 3.1, the systematic procedure of Sec. 2.3 is revisited with an eye on opportunities for mechanization. A prototype mechanical system for generating algorithms and routines is presented in Sec. 3.2. A number of examples are included, illustrating how to use the system. Sec. 3.3 comments on the scope and the limitations of the prototype mechanical systems.

Chapter 4 deals with the error analysis of algorithms obtained via the deriva-

tion methodology (Chapter 2) or generated by the prototype system (Chapter 3). Sec. 4.1 discusses extensions to the worksheet and the procedure first introduced in Chapter 2. Secs. 4.3, 4.4, and 4.5 present the stability analyses for operations of increasing complexity, with the intent of demonstrating the modularity of the methodology.

Finally, Chapter 5 summarizes the results of this thesis and proposes future research directions.

Chapter 2

Formal Derivation of Correct Linear Algebra Algorithms

As a first step towards a mechanical system, we describe a systematic procedure for deriving formally correct loop-based algorithms (see Section 1.2.3 for the definition of correctness). The input to the procedure is a mathematical specification of a target linear algebra operation, while the output is a family of correct loop-based algorithms that compute the operation. The development of such a procedure is important for two reasons: on one hand it allows a family of algorithms to be generated by following a sequence of unambiguous steps, and on the other hand it guarantees the formal correctness of the generated algorithms.

Proving the formal correctness of algorithms containing one or more loops has always been a challenging task: while the analysis requires a predicate that expresses a loop-invariant for each loop, there is no systematic way to discern a loop-invariant for an existing program. The burden is left on the developer, who must carefully state a predicate that is preserved during the execution of the loop. Our approach is different: as part of the procedure, a loop-invariant is identified *a priori*, and a loop that maintains such a predicate *true* through the computation is

constructed subsequently. In other words, the correctness of a given algorithm is not established *a posteriori*. Instead, constraints that a correct algorithm needs to satisfy are identified *a priori*, and statements that maintain these constraints are inserted.

E.W. Dijkstra in many occasions advised the computer sciences community on how to write programs and prove their correctness [14]:

> "The old technique was to make a program and then to subject it to a number of testcases where the answer was known; and when the testruns produced the correct result, this was taken as a sufficient ground for believing the program to be correct. [...] The most drastic discovery, however, was the last one, that what we then tried, viz. to prove the correctness of a given program, was in a sense putting the cart before the horse. A much more promising approach turned out to be letting correctness proof and program grow hand in hand: with the choice of the structure of the correctness proof one designs a program for which this proof is applicable. The fact that then the correctness concerns turn out to act as an inspiring heuristic guidance is an added benefit."

In this chapter we present a novel way of deriving algorithms for linear algebra operations.[1] It will become apparent that our research follows precisely Dijkstra's advice: our methodology consists in designing the structure of a proof of correctness first and building a program around such a proof later.

[1] The level of detail of the discussion is in preparation of the next chapter, where the steps will be made mechanical.

2.1 Correct Loop-Based Algorithms

"[...] no loop should be written down without
providing a proof for termination nor without stating
the relation whose invariance will not be destroyed
by the execution of the repeatable statement."
E.W. Dijkstra [13]. Turing Award lecture, 1972.

Our goal is to develop formally correct loop-based algorithms for computing a linear algebra operation $\mathcal{O}p$ starting solely from a mathematical description of the operation: central to the discussion is how to specify formally linear algebra operations. Depending on the level of abstraction and the formalism used, a given operation can be specified in a multitude of different ways; as an example we discuss the computation of the inverse function: the notation $y \leftarrow x^{-1}$ hints at the operation of inverting the input operand x, storing the result in the output variable y. Nonetheless such a notation does not provide specifics on the operands: it is not known whether x is a scalar or a matrix and no information about its domain nor regarding its size or structure or properties is given.

In order to generate formally correct algorithms, all the attributes for the input and output operands have to be fully specified. We require an input operation to be stated by means of three predicates: the *Precondition* (P_{pre}), the *Postcondition* (P_{post}) and the *Partitioned Matrix Expression* (PME). The precondition describes the domain, the dimensions and the properties of the input and output operands The postcondition defines the relations involving the input and output operands that hold *true* upon completion of the operation. Finally, the partitioned matrix expression declares how different parts of the solution can be computed in terms of parts of the input operands.

Example 1 (Cholesky) *The precondition and postcondition predicates for the*

Cholesky factorization $L = \Gamma(A)$ *are, respectively,*

$$P_{\text{pre}}: \quad \{ \;\; m(A) = n(A) = m(L) = n(L) \;\; \wedge \;\; A = \hat{A} \;\; \wedge$$
$$\text{SPD}(A) \;\; \wedge \;\; \text{LowerTriangular}(L) \;\; \wedge \;\; \text{Unknown}(L) \;\; \}$$

and

$$P_{\text{post}}: \{ \;\; LL^T = \hat{A} \;\; \wedge \;\; \text{Overwrite}(A, L) \;\; \}.$$

Here, predicate SPD(M) *is* true *iff* M *is a symmetric positive definite matrix, and predicate* LowerTriangular(M) *returns* true *iff* M *is a square lower triangular matrix.*

The predicate Unknown$(Z_1, Z_2, \ldots, Z_i)$ indicates that the matrices $Z_1, Z_2, \ldots, Z_i$ are unknown variables, i.e., they represent the quantities that we want to compute.
If Z is both an input and output variable, i.e., it is to be overwritten during the computation, we use the notation $\hat{Z}$ to denote the initial contents of Z.
The predicate Overwrite$(Z_{\text{out}}, Z_{\text{in}})$ indicates that variable Z_{in} overwrites variable Z_{out}.
The functions $m(Z)$ and $n(Z)$ return the number of rows and columns of matrix Z, respectively. Alternatively, $Size(Z)$ is a function that returns the couple $m(Z) \times n(Z)$.

The partitioned matrix expression is

$$\text{PME} \;\; : \;\; \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c} \Gamma(\hat{A}_{TL}) & 0 \\ \hline \hat{A}_{BL} L_{TL}^{-T} & \Gamma(\hat{A}_{BR} - L_{BL} L_{BL}^T) \end{array}\right) \;\; \wedge$$
$$Size(L_{TL}) = Size(\hat{A}_{TL}) = k \times k, \;\; k \in [0, m(L)].$$

When partitioning a matrix, the subscript letters T, B, L and R signify Top, Bottom, Left and Right, respectively.

This PME indicates that independently of the partitioning size, the Top Left quadrant of the solution matrix L contains the Cholesky factor of the corresponding quadrant of matrix $\hat{A}$: $L_{TL} = \Gamma(\hat{A}_{TL})$. Similarly, the Bottom Left and Bottom Right quadrants of L contain respectively $\hat{A}_{BL}L_{TL}^{-T}$ and the Cholesky factor of matrix $\hat{A}_{BR} - L_{BL}L_{BL}^T$. These three relations have to be satisfied at the same time for a matrix L to be the Cholesky factor of A.

Example 2 (Triangular Inverse) *The operation that computes the inverse of a lower triangular matrix L overwriting the input matrix is specified by the predicates:*

$$
\begin{aligned}
P_{\text{pre}} \quad &: \quad \{\ Size(L) = n \times n \ \wedge\ \text{LowerTriangular}(L) \ \wedge \\
&\qquad L = \hat{L} \ \wedge\ \text{Unknown}(L) \ \wedge\ \text{Inv}(L)\ \}, \\
P_{\text{post}} \quad &: \quad \{\ L = \hat{L}^{-1}\ \}, \\
\text{PME} \quad &: \quad \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c} \hat{L}_{TL}^{-1} & 0 \\ \hline -\hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1} & \hat{L}_{BR}^{-1} \end{array}\right) \ \wedge \\
&\qquad Size(L_{TL}) = Size(\hat{L}_{TL}) = k \times k,\ k \in [0, m(L)].
\end{aligned}
$$

The predicate $\text{Inv}(L)$ *is true iff the matrix L is invertible.*

In the rest of the thesis, in order to remove clutter from the exposition, we omit the clauses indicating the size and the shape of the operands whenever this information is obvious from the context.

The specification of a target linear algebra operation $\mathcal{O}p$ by means of the precondition and postcondition is a clear preparation for using Hoare's triples for designing a proof of formal correctness. Knowing the predicates P_{pre} and P_{post}, the goal is to construct an algorithm $\mathcal{A}$ such that the Hoare's triple

$$\{P_{\text{pre}}\}\ \mathcal{A}\ \{P_{\text{post}}\} \tag{2.1}$$

is satisfied. If such an algorithm $\mathcal{A}$ can be found, then it will correctly compute the operation $\mathcal{O}p$, by construction.

We restrict ourselves to algorithms that consist of a simple initialization followed by a loop. Such a restriction is not as severe as it may appear at first glance. As mentioned in the introduction, the numerical linear algebra community has made tremendous strides towards modularity and, as a consequence, almost any linear algebra operation can be decomposed into more basic operations (building blocks) that are meaningful linear algebra operations themselves and exhibit this simple structure. In Section 2.6 we speculate on how our framework also covers recursive algorithms and why loop-based algorithms are to be preferred for dense linear algebra computations.

Expression 2.1 can be rewritten to take into account that the algorithm $\mathcal{A}$ can be written as $\mathcal{A}1; \mathcal{A}2$, where $\mathcal{A}1$ is an initialization statement and $\mathcal{A}2$ is a loop:

$$\{P_{\text{pre}}\}\ \mathcal{A}1;\ \mathcal{A}2\ \{P_{\text{post}}\}.$$

The Floyd-Hoare logic tells us that this triple is satisfied if the following two triples are in turn satisfied

$$\{P_{\text{pre}}\}\ \mathcal{A}1\ \{Q\}, \qquad \{Q\}\ \mathcal{A}2\ \{P_{\text{post}}\},$$

for a suitable state Q. Let us now substitute $\mathcal{A}1$ with the more intuitive label Init and $\mathcal{A}2$ with the explicit loop structure **While** G **do** *Body* **end**, where G is the loop-guard and *Body* denotes the computation executed at each iteration of the loop. The two triples become:

$$\{P_{\text{pre}}\}\ \text{Init}\ \{Q\}, \qquad \{Q\}\ \textbf{While}\ G\ \textbf{do}\ \mathit{Body}\ \textbf{end}\ \{P_{\text{post}}\}. \tag{2.2}$$

The reader may have the feeling that because of these transformations we are now

in a more complicated situation than we were at the beginning: the state Q, representing the postcondition for the initialization ($\mathcal{A}1$) and the precondition for the while loop ($\mathcal{A}2$) is unknown. We are no longer in the pleasant situation where both the precondition and the postcondition of a Hoare triple are known. The "Fundamental Invariance Theorem for Loops" comes to our rescue; here we paraphrase the theorem from Gries and Schneider' book *A Logical Approach to Discrete Math* [22].

> This theorem refers to an assertion P_{inv} that holds before and after each iteration (provided it holds before the first). Such a predicate is called a *loop-invariant.*
>
> (12.43) **Fundamental Invariance Theorem.** Suppose
>
> 1. $\{P_{\text{inv}} \wedge G\}\ S\ \{P_{\text{inv}}\}$ holds – i.e., execution of S begun in a state in which P_{inv} and G are *true* terminates with P_{inv} *true* – and
> 2. $\{P_{\text{inv}}\}$ **while** G **do** S **end** $\{true\}$ – i.e., execution of the loop begun in a state in which P_{inv} is *true* terminates.
>
> Then $\{P_{\text{inv}}\}$ **while** G **do** S **end** $\{P_{\text{inv}} \wedge \neg G\}$ holds. In other words, if the loop is entered in a state where P_{inv} is *true*, it will complete in a state where P_{inv} is *true* and guard G is *false*.

The text proceeds to prove this theorem using the axiom of mathematical induction. The theorem tells us that if we could 1) discover a loop-invariant P_{inv} for our loop $\mathcal{A}2$, and 2) prove the termination of such a loop, then the following Hoare's triple would hold *true*:

$$\{P_{\text{inv}}\}\ \textbf{While}\ G\ \textbf{do}\ \text{Body}\ \textbf{end}\ \{P_{\text{inv}} \wedge \neg G\}. \tag{2.3}$$

The comparison of Expressions (2.2) (what we desire) and (2.3) (what the theorem ensures) leads us to the following consideration: if the predicate $\{P_{\text{inv}} \wedge \neg G\}$ implies

the postcondition P_{post}, then the natural choice for the state Q is P_{inv} (note that a loop-invariant is *true* before the loop is ever entered); this choice brings us again to the situation of dealing with Hoare triples whose precondition and postcondition are both known. The triples 2.2 can be rewritten as

$$\{P_{\text{pre}}\}\ \text{Init}\ \{P_{\text{inv}}\}, \qquad \{P_{\text{inv}}\}\ \textbf{While}\ G\ \textbf{do}\ \text{Body}\ \textbf{end}\ \{\neg G \wedge P_{\text{inv}}\}; \tag{2.4}$$

the problem of finding statements Init, G and Body satisfying these two triples is now equivalent to building a correct algorithm $\mathcal{A}$ that computes $\mathcal{O}p$.

Before looking more closely at the anatomy of the three statements (Init, G and Body) composing $\mathcal{A}$, we list the assumptions made so far:

1. The predicate P_{inv} is a loop-invariant for loop $\mathcal{A}2$;
2. The loop $\mathcal{A}2$, when executed in a state in which P_{inv} is *true*, terminates;
3. The predicate $\{\neg G \wedge P_{\text{inv}}\}$ implies P_{post}.

A thorough discussion on how to state a loop-invariant for the loop $\mathcal{A}2$ *a priori* is deferred to Section 2.5; for now we simply state that a loop-invariant can be deduced from the PME for the operation $\mathcal{O}p$. In order to ensure the termination of the loop, we require the loop-guard G to measure the advancement in the traversal of the operands and the loop-body to include statements to make progress for traversing the operands: each iteration contains a first step in which one or more **Repartition** statements expose new submatrices and subvectors of the operands, a second step of numerical computations and a third and last step in which **Continue with** statements re-assemble submatrices and subvectors to guarantee progress. This simple structure was already featured in the algorithm that we presented for computing the Cholesky factor in Fig. 1.4 (a traditional derivation of this algorithm is given in Appendix A). In that algorithm, the initialization Init is

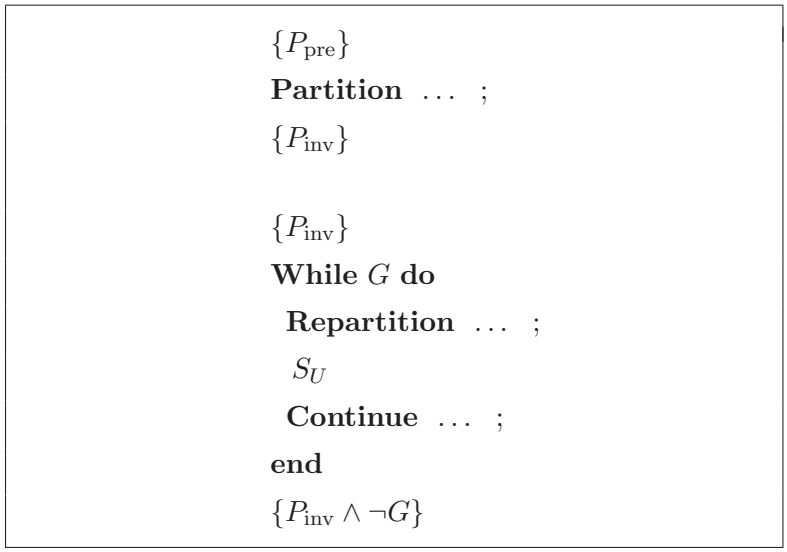

Figure 2.1: Template for a formal proof of correctness for linear algebra algorithms consisting of an initialization step followed by a loop.

a simple partitioning of the program variable (matrix A), in which no actual computation performed. The loop-guard G is a comparison of the size of two matrices (A and A_{TL}) and it is chosen so that $\{\neg G \wedge P_{\text{inv}}\}$ implies the postcondition P_{post}. Finally, the loop-body consists of one first statement to repartition the matrix A, a number of mathematical instructions and one last statement to reassemble A which ensures progress towards making G *false*.

Let us now incorporate the facts on the nature of the statements Init, G and Body into Expression 2.4. The result is the skeleton of a proof of correctness for a linear algebra algorithm (Fig 2.1). In the next sections we describe how to use such an infrastructure to derive algorithms computing a target operation.

Step	Annotated Algorithm: $[D, E, F, \ldots] = \text{op}(A, B, C, D, \ldots)$
1a	$\{\ P_{\text{pre}}\ \}$
3	**Partition** $\quad$ **where**
2	$\{\ P_{\text{inv}}\ \}$
4	**While** G **do**
2,4	$\{(\ P_{\text{inv}}\ ) \wedge (\ G\ )\}$
5a	**Repartition** $\quad$ **where**
6	$\{\ P_{\text{before}}\ \}$
8	S_U
7	$\{\ P_{\text{after}}\ \}$
5b	**Continue with**
2	$\{\ P_{\text{inv}}\ \}$
	endwhile
2,4	$\{(\ P_{\text{inv}}\ ) \wedge \neg (\ G\ )\}$
1b	$\{\ P_{\text{post}}\ \}$

Figure 2.2: Worksheet for developing linear algebra algorithms.

2.2 Worksheet

> "[...] the programmer should let correctness proof and program grow hand in hand. [...] If one first asks oneself what the structure of a convincing proof would be and, having found this, then constructs a program satisfying this proof's requirements, then these correctness concerns turn out to be a very effective heuristic guidance."
>
> E.W. Dijkstra [13]. Turing Award lecture, 1972.

Figure 2.2 shows a generic "worksheet" for deriving a large class of linear algebra algorithms. The worksheet resembles the skeleton of a linear algebra algorithm in the sense that it consists of an initialization step (Step 3) followed by a **While** loop (Step 4) inside which operations on the input/output variables are performed

(Steps 5a, 8 and 5b); the gray-shaded boxes contain statements that would appear in actual code. Such a skeleton is enriched with predicates in curly brackets, expressing the status of input/output variables at different stages in the algorithm (Steps 1a, 2, 2,4, 6, 7, 1b). The numbers in the "Step" column refer to the order in which the worksheet is filled to generate an algorithm.

Given a target operation $\mathcal{O}p$ specified by the precondition P_{pre} and post-condition P_{post} (and the PME), in the last section we have reduced the problem of finding an algorithm $\mathcal{A}$ for $\mathcal{O}p$ satisfying the triple $\{P_{\text{pre}}\}\ \mathcal{A}\ \{P_{\text{post}}\}$ to the problem of making the two triples in (2.4) *true* by finding a loop-invariant P_{inv} and

1. An initialization statement Init;
2. A loop-guard G involving comparisons on the size of input/output operands;
3. Statements **Repartition** and **Continue with** (part of the Body of the **While** loop), responsible of 1) traversing input/output operands, and 2) ensuring progress towards rendering the loop-guard *false*;
4. A set of computational statements with the property of maintaining a loop-invariant P_{inv}.

The worksheet in Fig. 2.2 exhibits the same breakdown of the problem. Lines 1a and 1b contain the predicates P_{pre} and P_{post} respectively. The first triple from (2.4), $\{P_{\text{pre}}\}$ Init $\{P_{\text{inv}}\}$, is captured by the first three lines (Steps 1a, 3, 2), while the second triple from 2.4 is captured in lines from the third to the next to last one (from Step 2 to Step 2,4). There the loop-body is decomposed into three separate statements (Steps 5a, 8, 5b) that are interleaved with four predicates in curly brackets (Steps 2,4 6, 7, 2). Steps 2,4 and 2 respectively indicate that the loop-invariant holds *true* at the beginning and the end of each loop iteration, while P_{before} and P_{after} at Steps 6 and 7 represent P_{inv} in terms of the repartitioned operands, that is, before and after the computational statements of Step 8.

The worksheet in Fig. 2.2 represents the "structure of a convincing proof;" thanks to this framework, the user makes "the correctness proof and the program grow hand in hand" by filling in the predicates and the statements. The idea is that if it is possible to state the assertion in curly brackets first, then the statements can be chosen so that those assertions hold *true* at the indicated points in the algorithm.

In the next section we give an 8-step procedure, "the Recipe," that dictates how to derive loop-invariant (Step 2), initialization (Step 3), loop-guard (Step 4), statements to sweep through the operands (Steps 5a and 5b) and computational statements (Step 8) for an algorithm once P_{pre}, P_{post} and PME of a target operation are given. Since all the statements are derived to satisfy Hoare triples, the final algorithm is guaranteed to be formally correct.

2.3 An Eight-Step Recipe

Eight steps are typically sufficient to transition from the mathematical specification of a target linear algebra operation to an algorithm that computes the operation. We start presenting each step, with reference to Fig. 2.2, pointing out how formal derivation techniques are used to relate predicates and statements. At the end of the treatment it will become clear that the process of filling in the worksheet relies entirely on the ability to determine loop-invariants. We discuss how to derive loop-invariants in Sec. 2.5. Sec. 2.4 contains a complete example that derives algorithms for the computation of the inverse of a triangular matrix.

1a,1b **Determine P_{pre} and P_{post}:** A generic target operation is indicated by $[D, E, F, \ldots] = \text{op}(A, B, C, D, \ldots)$. Some operands may be overwritten; when this is the case, we use the ˆ-notation to indicate the original contents of a variable. Predicates P_{pre}, the precondition, and P_{post}, the postcondition, respectively describe constraints and properties of the operands, and the desired

state upon completion of the algorithm. P_{pre}, P_{post} and PME are the specification of the target operation and they represent the inputs for the process of deriving N algorithms $\{\mathcal{A}_i\}_{i=1}^{N}$, such that the triple $\{P_{\text{pre}}\}\ \mathcal{A}_i\ \{P_{\text{post}}\}$ is satisfied.

2 **Determine loop-invariant** P_{inv}: The loop-invariant is a predicate that expresses the state of input and output variables during the execution of the loop and it holds true throughout the entire computation. Being able to find a loop-invariant means that the first hypothesis of the Fundamental Invariance Theorem [22] is satisfied: $\{P_{\text{inv}} \wedge G\}\ S\ \{P_{\text{inv}}\}$ (see Section 2.1). A posteriori automatic detection of loop-invariants is a problem which has been deeply investigated, with results far from being generally applicable. We will show that for a class of linear algebra operations it is possible to identify methodically acceptable loop-invariants *a priori* (see Section 2.5).

3 **Determine initialization**: Since the loop-invariant holds *true* before the loop commences, the initialization Init, Step 3 in Fig. 2.2, must have the property that starting in the state P_{pre}, it sets the variables to a state in which P_{inv} holds: $\{P_{\text{pre}}\}$ Init $\{P_{\text{inv}}\}$. We require the initialization to consist of a (possibly empty) list of statements to partition the operands or to set them to specific values.

4 **Determine loop-guard G**: The loop-guard is the condition under which the program enters the loop; when the loop completes $\neg G$ holds *true*. If the loop terminates, then also the second hypothesis of the Fundamental Invariance Theorem is satisfied, hence the thesis of the theorem can be asserted: $(P_{\text{inv}} \wedge \neg G)$ holds *true* after the loop. Note that if $(P_{\text{inv}} \wedge \neg G)$ implies P_{post} then we can conclude that, upon termination, the loop correctly computes P_{post}. An appropriate loop-guard can be derived comparing the loop-invariant

P_{inv} and the postcondition P_{post}.

5a,5b **Determine how to march through the operands**: Step 5a and 5b are responsible to guarantee termination of the loop by traversing the operands. Step 5a exposes new regions of the operands and Step 5b re-assembles the regions so that progress is achieved. In terms of a Hoare triple, these two statements are chosen so that $\{P_{\text{inv}} \wedge G\}$ **while G do** Step 5a; Step 5b; **end** $\{true\}$ is satisfied. No actual computation happens at these stages, they merely represent indexing operations.

6 **Determine** P_{before}: We already determined the statement for traversing the operands (Step 5a); predicate P_{before} expresses P_{inv} after the repartitioning takes place. Formally: $\{P_{\text{inv}}\}$ Step 5a $\{P_{\text{before}}\}$.

7 **Determine** P_{after}: Similar to the previous step, this predicate only expresses P_{inv} in terms of partitioned operands: it must hold $\{P_{\text{after}}\}$ Step 5b $\{P_{\text{inv}}\}$, therefore the predicate P_{after} denotes the loop-invariant before the **Continue with** (Step 5b) execution.

8 **Determine the updates** $\mathbf{S}_U$: The computational statements $\mathbf{S}_U$ must be such that the triple $\{P_{\text{before}}\}$ $\mathbf{S}_U$ $\{P_{\text{after}}\}$ holds *true*.

2.4 Example: Inverting a Triangular Matrix

We show how to apply the eight steps presented in Section 2.3 to derive algorithms for a specific target operation. The operation we consider is the inversion of a triangular matrix $L := L^{-1}$ where L is an $m \times m$ lower triangular matrix.

1 **Determine** P_{pre} **and** P_{post}**.** In Example 2 we already discussed how to formally specify this operation by means of the predicates

Step	Annotated Algorithm: $L := L^{-1}$
1a	$\left\{ L = \hat{L} \right\}$
3	**Partition** $L = \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$ and $\hat{L} = \left(\begin{array}{c\|c} \hat{L}_{TL} & 0 \\ \hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right)$ **where** L_{TL} and $\hat{L}_{TL}$ are 0×0
2	$\left\{ \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL}^{-1} & 0 \\ \hline -L_{BL}L_{TL}^{-1} & L_{BR} \end{array}\right) \right\}$
4	**While** $\neg\text{SameSize}(L, L_{TL})$ **do**
2,4	$\left\{ \left(\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL}^{-1} & 0 \\ \hline -L_{BL}L_{TL}^{-1} & L_{BR} \end{array}\right) \right) \wedge (\neg\text{SameSize}(L, L_{TL})) \right\}$
5a	**Repartition** $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \ldots$ **where** L_{11} and $\hat{L}_{11}$ are $b \times b$
6	$\left\{ \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c\|c\|c} \hat{L}_{00}^{-1} & 0 & 0 \\ \hline -\hat{L}_{10}\hat{L}_{00}^{-1} & \hat{L}_{11} & 0 \\ \hline -\hat{L}_{20}\hat{L}_{00}^{-1} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right) \right\}$
8	$L_{21} := -L_{21}\ L_{11}^{-1}$ (TRSM) $L_{20} := L_{20} + L_{21}\ L_{10}$ (GEMM) $L_{10} := L_{11}^{-1}\ L_{10}$ (TRSM) $L_{11} := L_{11}^{-1}$ (Matrix Inversion)
7	$\left\{ \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c\|c\|c} \hat{L}_{00}^{-1} & 0 & 0 \\ \hline -\hat{L}_{11}^{-1}\hat{L}_{10}\hat{L}_{00}^{-1} & \hat{L}_{11}^{-1} & 0 \\ \hline -\hat{L}_{20}L_{00}^{-1} - \hat{L}_{21}\hat{L}_{11}^{-1}\hat{L}_{10}\hat{L}_{00}^{-1} & -\hat{L}_{21}\hat{L}_{11}^{-1} & \hat{L}_{22} \end{array}\right) \right\}$
5b	**Continue with** $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \ldots$
2	$\left\{ \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL}^{-1} & 0 \\ \hline -L_{BL}L_{TL}^{-1} & L_{BR} \end{array}\right) \right\}$
	endwhile
2,4	$\left\{ \left(\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL}^{-1} & 0 \\ \hline -L_{BL}L_{TL}^{-1} & L_{BR} \end{array}\right) \right) \wedge \neg (\neg\text{SameSize}(L, L_{TL})) \right\}$
1b	$\left\{ L = \hat{L}^{-1} \right\}$

Figure 2.3: Worksheet completed for the inversion of a triangular matrix.

P_{pre} : { $Size(L) = m \times m \ \wedge$ LowerTriangular(L) $\wedge$ $L = \hat{L} \ \wedge$
Unknown(L) $\wedge$ Inv(L) }, and
$P_{\text{post}} : \{L = \hat{L}^{-1}\}$.

2 **Determine loop-invariant P_{inv}.** The partitioned matrix expression for this operation is (from Example 2):

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c} \hat{L}_{TL}^{-1} & 0 \\ \hline -\hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1} & \hat{L}_{BR}^{-1} \end{array}\right). \tag{2.5}$$

This equation expresses how the quadrants of the output matrix L are functions of the quadrants of the input matrix $\hat{L}$. In other words the PME provides a formula for

$$\left(\begin{array}{c|c} \hat{L}_{TL} & 0 \\ \hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right)^{-1}$$

in terms of the quadrants $\hat{L}_{TL}$, $\hat{L}_{BL}$, $\hat{L}_{BR}$. Let us assume there exists a loop-based algorithm that correctly computes the inverse of a matrix $\hat{L}$; if the execution of such an algorithm is interrupted at an intermediate stage (at the beginning of an iteration) and the matrix L that is being computed is inspected, we observe that only a subset of the operations appearing in Equation 2.5 have been performed (otherwise the computation would have terminated). Different loop-invariants are obtained by selecting different subsets of the operations contributing to the final result; a partial list of possible loop invariants for this operations is given in Table 2.1. In order to have a complete list of loop-invariants one needs to consider also the ones progressing from the BR to the TL corner, such as

$$\left(\begin{array}{c|c} L_{TL} = L_{TL} & 0 \\ \hline L_{BL} = -\hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1} & L_{BR} = \hat{L}_{BR}^{-1} \end{array}\right),$$

#	Loop − invariant
1	$\left(\begin{array}{c\|c} L_{TL} = \hat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} = \hat{L}_{BL} & L_{BR} = \hat{L}_{BR} \end{array}\right)$
2	$\left(\begin{array}{c\|c} L_{TL} = \hat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} = -\hat{L}_{BL}\hat{L}_{TL}^{-1} & L_{BR} = \hat{L}_{BR} \end{array}\right)$
3	$\left(\begin{array}{c\|c} L_{TL} = \hat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} = -\hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1} & L_{BR} = \hat{L}_{BR} \end{array}\right)$
4	$\left(\begin{array}{c\|c} L_{TL} = \hat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} = -\hat{L}_{BR}^{-1}\hat{L}_{BL} & L_{BR} = \hat{L}_{BR} \end{array}\right)$

Table 2.1: Loop-invariants for computing the inverse of a triangular matrix progressing from the TL corner.

which corresponds to an algorithm that computes L one (or more) column at a time, proceeding from the bottom of matrix L towards the top.

In the remainder of this section we concentrate on the following loop-invariant

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c} \hat{L}_{TL}^{-1} & 0 \\ \hline -\hat{L}_{BL}\hat{L}_{TL}^{-1} & \hat{L}_{BR} \end{array}\right),$$

which becomes P_{inv} in the worksheet of Fig. 2.2 as illustrated in Fig. 2.3.

3 **Determine initialization.** The loop-invariant P_{inv} and the precondition P_{pre} dictate the initialization. $\{P_{\text{pre}}\}$ Init $\{P_{\text{inv}}\}$ will be made *true* by means of partitionings (indexing) only. We have:

$$\{L = \hat{L}\} \text{ Init } \left\{ \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c} \hat{L}_{TL}^{-1} & 0 \\ \hline -\hat{L}_{BL}\hat{L}_{TL}^{-1} & \hat{L}_{BR} \end{array}\right) \right\}.$$

The initialization statements $L = \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$ and $\hat{L} = \left(\begin{array}{c|c} \hat{L}_{TL} & 0 \\ \hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right)$, with L_{TL} and $\hat{L}_{TL}$ 0×0 quadrants, require no computation and make the Hoare triple *true*.

If no initialization can be found so that $\{P_{\text{pre}}\}$ Init $\{P_{\text{inv}}\}$ is *true* then the loop-invariant is declared infeasible.

4 **Determine loop-guard G.** Upon completion of the loop, the predicate $P_{\text{inv}} \wedge \neg G$ holds *true* (Fundamental Invariance Theorem). Thus, by choosing a loop-guard G such that $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$, we guarantee that if the loop completes, it does so in a state that implies that the desired result has been computed. In this example we are looking for G such that:

$$\left(\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c} \hat{L}_{TL}^{-1} & 0 \\ \hline -\hat{L}_{BL}\hat{L}_{TL}^{-1} & \hat{L}_{BR} \end{array}\right) \wedge G\right) \Rightarrow (L = \hat{L}^{-1}).$$

If the quadrant L_{TL} encompasses the entire matrix L, it is easy to see that the previous implication is established. Hence we choose G to be $\neg\text{SameSize}(L, L_{TL})$, as illustrated in Step 4 of the worksheet in Fig. 2.3. The function $\text{SameSize}(L, L_{TL})$ returns *true* iff the dimensions of matrices L and L_{TL} are equal.

If no loop-guard can be found so that $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$, then the loop-invariant is declared infeasible.

5 **Determine how to march through the operands.** The loop-guard G and the initialization step dictate in what direction the variables need to be repartitioned to make progress towards making the predicate G *false*.

Once it is known what quadrant has to eventually encompass the whole matrix, the approach is to expose new parts of the matrix at the top of the loop (Step

5a) and then move them to the appropriate regions at the bottom of the loop (Step 5b), as illustrated in Fig. 2.3.

The initialization step sets L_{TL} to be 0×0, while upon completion of the loop, quadrant L_{TL} has to encompass the complete matrix. Thus, the boundaries indicating where the computation reached so far, denoted by the thick lines, must be moved forward as part of the body of the loop with the goal of augmenting the TL quadrant. This is achieved by exposing new parts from the quadrants L_{BL} and L_{BR} first and adding them to L_{TL} subsequently. The statements are respectively

Repartition

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} \hat{L}_{TL} & 0 \\ \hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \hat{L}_{00} & 0 & 0 \\ \hline \hat{L}_{10} & \hat{L}_{11} & 0 \\ \hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right)$$

where L_{11} and $\hat{L}_{11}$ are $b \times b$

and

Continue with

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} \hat{L}_{TL} & 0 \\ \hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} \hat{L}_{00} & 0 & 0 \\ \hline \hat{L}_{10} & \hat{L}_{11} & 0 \\ \hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right).$$

The quantity b is the algorithmic block size: if $b = 1$, the algorithm proceeds exposing one new column and row per iteration, i.e., it is "unblocked." By contrast, if $b > 1$, the algorithm is "blocked," meaning that it operates on several rows and columns at one time. When b is chosen to be greater or equal to half the size of one of the operands (in this case L), the algorithm is purely recursive (see Section 2.6).

6 **Determine P_{before}.** The state P_{before} represents the loop-invariant when the repartitionings determined in Step 5a have been applied. Executing the repartitioning statement corresponds to renaming or, equivalently, to performing textual substitution. Mathematically the operation for determining P_{before} is

$$\textbf{Simplify}\left(P_{\text{inv}}\Big|_{\textbf{Repartitioning}} \right),$$

where the vertical bar signifies the application of textual substitution rules. The **Simplify** function is necessary in order to derive an explicit formula for the loop-invariant as function of the newly exposed regions. This is achievable via algebraic manipulations and known mathematical relations.

The repartition statement for the matrix $\hat{L}$ may be encoded by the following three substitution rules:

$$\hat{L}_{TL} \rightarrow \hat{L}_{00}, \quad \hat{L}_{BL} \rightarrow \left(\begin{array}{c} \hat{L}_{10} \\ \hline \hat{L}_{20} \end{array} \right), \quad \hat{L}_{BR} \rightarrow \left(\begin{array}{c|c} \hat{L}_{11} & 0 \\ \hline \hat{L}_{21} & \hat{L}_{22} \end{array} \right).$$

Three similar substitution rules define the repartition statement for matrix L. The application of such rules to loop-invariant

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right) = \left(\begin{array}{c|c} \hat{L}_{TL}^{-1} & 0 \\ \hline -\hat{L}_{BL}\hat{L}_{TL}^{-1} & \hat{L}_{BR} \end{array} \right),$$

yields $(P_{\text{inv}}\Big|_{\textbf{Repartitioning}})$

$$\left(\begin{array}{c|c} L_{00} & 0 \\ \hline \left(\begin{array}{c} L_{10} \\ \hline L_{20} \end{array} \right) & \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array} \right) \end{array} \right) = \left(\begin{array}{c|c} \hat{L}_{00}^{-1} & 0 \\ \hline -\left(\begin{array}{c} \hat{L}_{10} \\ \hline \hat{L}_{20} \end{array} \right) \hat{L}_{00}^{-1} & \left(\begin{array}{c|c} \hat{L}_{11} & 0 \\ \hline \hat{L}_{21} & \hat{L}_{22} \end{array} \right) \end{array} \right),$$

and the function **Simplify** returns

$$\left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c|c|c} \hat{L}_{00}^{-1} & 0 & 0 \\ \hline -\hat{L}_{10}\hat{L}_{00}^{-1} & \hat{L}_{11} & 0 \\ \hline -\hat{L}_{20}\hat{L}_{00}^{-1} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right)$$

which is the desired state P_{before}.

7 **Determine P_{after}.**

The predicate P_{after} is such that upon completion of the **Continue** statement P_{inv} holds *true*. **Continue with** only manipulates regions of the operands. It corresponds to a set of textual substitution rules from the set of regions exposed by the **Repartition** statement into the quadrants defined by the initialization step.

Since P_{inv} is known, the idea is to determine P_{after} starting from the expression of the loop-invariant and executing the converse of the **Continue** statement, i.e., executing the textual substitution rules backwards. Mathematically, P_{after} is determined by

$$\textbf{Simplify}\left(P_{\text{inv}}\Big|_{\textbf{Continue}^{-1}}\right).$$

The statement

Continue with

$$\left(\begin{array}{c|c} \hat{L}_{TL} & 0 \\ \hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} \hat{L}_{00} & 0 & 0 \\ \hline \hat{L}_{10} & \hat{L}_{11} & 0 \\ \hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right)$$

is equivalent to the textual substitution rules

$$\left(\begin{array}{c|c} \hat{L}_{00} & 0 \\ \hline \hat{L}_{10} & \hat{L}_{11} \end{array}\right) \to \hat{L}_{TL}, \quad \left(\begin{array}{c|c} \hat{L}_{20} & \hat{L}_{21} \end{array}\right) \to \hat{L}_{BL}, \quad \hat{L}_{22} \to \hat{L}_{BR}.$$

Therefore the rules for $\mathbf{Continue}^{-1}$ are:

$$\hat{L}_{TL} \to \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right), \quad \hat{L}_{BL} \to \left(\begin{array}{c|c} L_{20} & L_{21} \end{array}\right), \quad \hat{L}_{BR} \to \hat{L}_{22}.$$

Three similar substitution rules define the **Continue** statement for matrix L. Now $P_{\text{inv}}\Big|_{\mathbf{Continue}^{-1}}$ translates into

$$\left(\begin{array}{c|c} \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right) & 0 \\ \hline \left(\begin{array}{c|c} L_{20} & L_{21} \end{array}\right) & L_{22} \end{array}\right) = \left(\begin{array}{c|c} \left(\begin{array}{c|c} \hat{L}_{00} & 0 \\ \hline \hat{L}_{10} & \hat{L}_{11} \end{array}\right)^{-1} & 0 \\ \hline -\left(\begin{array}{c|c} \hat{L}_{20} & \hat{L}_{21} \end{array}\right)\left(\begin{array}{c|c} \hat{L}_{00} & 0 \\ \hline \hat{L}_{10} & \hat{L}_{11} \end{array}\right)^{-1} & \hat{L}_{22} \end{array}\right),$$

and simplifying (**Simplify**)

$$\left(\begin{array}{cc|c} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{cc|c} \hat{L}_{00}^{-1} & 0 & 0 \\ -\hat{L}_{11}^{-1}\hat{L}_{10}\hat{L}_{00}^{-1} & \hat{L}_{11}^{-1} & 0 \\ \hline -\hat{L}_{20}L_{00}^{-1} - \hat{L}_{21}\hat{L}_{11}^{-1}\hat{L}_{10}\hat{L}_{00}^{-1} & -\hat{L}_{21}\hat{L}_{11}^{-1} & \hat{L}_{22} \end{array}\right)$$

which is the desired state P_{after}.

8 **Determine the updates $\mathbf{S}_U$.**

By comparing the state in Step 6 with the desired state in Step 7 the required update, the statements S_U given in Step 8, are determined.

The contents of matrix L in the states P_{before}

$$\left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c|c|c} \hat{L}_{00}^{-1} & 0 & 0 \\ \hline -\hat{L}_{10}\hat{L}_{00}^{-1} & \hat{L}_{11} & 0 \\ \hline -\hat{L}_{20}\hat{L}_{00}^{-1} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right)$$

and P_{after}

$$\left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c|c|c} \hat{L}_{00}^{-1} & 0 & 0 \\ \hline -\hat{L}_{11}^{-1}\hat{L}_{10}\hat{L}_{00}^{-1} & \hat{L}_{11}^{-1} & 0 \\ \hline -\hat{L}_{20}L_{00}^{-1} - \hat{L}_{21}\hat{L}_{11}^{-1}\hat{L}_{10}\hat{L}_{00}^{-1} & -\hat{L}_{21}\hat{L}_{11}^{-1} & \hat{L}_{22} \end{array}\right)$$

differ in the submatrices $(1,0)$, $(1,1)$, $(2,0)$, and $(2,1)$. The updates $\mathbf{S}_U$ needed to ensure that the triple $\{P_{\text{before}}\}\ S_U\ \{P_{\text{after}}\}$ is verified are

$$\begin{aligned} L_{21} &:= -L_{21}L_{11}^{-1} \\ L_{20} &:= L_{20} + L_{21}L_{10} \\ L_{10} &:= L_{11}^{-1}L_{10} \\ L_{11} &:= L_{11}^{-1} \end{aligned}$$

The worksheet completed for the inversion of a triangular matrix is shown in Fig. 2.3 while Fig. 2.4 (left) presents the final version of the algorithm, cleaned of the assertions within curly-brackets. Notice that the last computational update, $L_{11} := L_{11}^{-1}$, is a triangular matrix inversion itself and can be solved recursively or by means of an unblocked algorithm. Such an algorithm can be easily derived using the worksheet (Fig. 2.3) and forcing b to be 1 in Step 5. The right part of Fig. 2.4 illustrates the resulting algorithm.

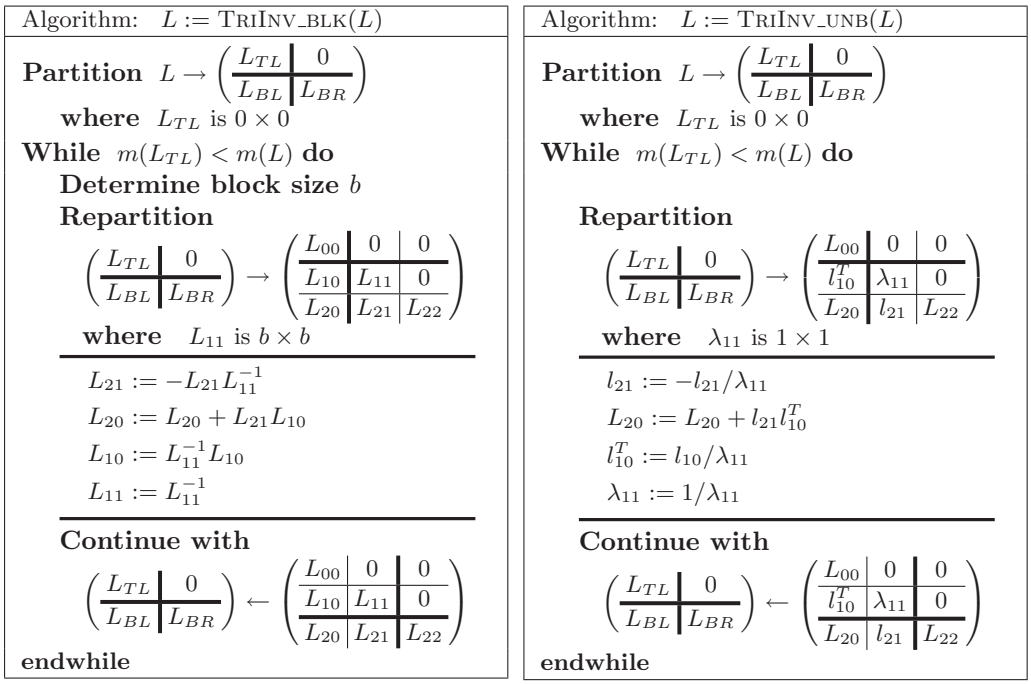

Figure 2.4: Blocked (on the left) and unblocked (on the right) algorithms for computing the inverse of a triangular matrix.

2.5 Partitioned Matrix Expression (PME) and Loop-Invariants

Together with P_{pre} and P_{post}, the PME is a predicate that specifies a target linear algebra operation. Its role is to express different parts of the output operands in terms of parts of the input operands. In this section we discuss how to derive a PME and the relation between PME and loop-invariants. Since the process of filling in the worksheet in Fig. 2.2 depends entirely on the loop-invariant, and different loop-invariants are deduced from the PME, it is this predicate that prescribes the generation of a family of algorithms.

Examples 1 and 2 from Section 2.1 report the PMEs for the Cholesky factorization and for the triangular inversion, respectively. In this section we illustrate the

steps to derive the PME for the Cholesky factorization (with the knowledge of P_{pre} and P_{post}) and how to obtain a list a loop-invariants from the PME. The treatment of the inversion of a triangular matrix would involve analogous steps. Finally we turn our attention to a considerably more complex problem, the triangular discrete time Sylvester equation, which will serve as an example of an operation that admits multiple PMEs.

Since we want to determine how different quadrants of the input operands contribute to the computation of the output operands, we start by picking one of the operands and by partitioning it into two submatrices, either horizontally or vertically, or into four quadrants. The partitioning corresponds to the assumption that algorithms progress through data in a systematic fashion. A rule of thumb is that if a matrix has a special structure, e.g., triangular or symmetric, it is partitioned into quadrants that are consistent with the structure. If the matrix has no special structure, it can be partitioned vertically, horizontally, or into quadrants.

2.5.1 Example: Cholesky Factorization

In the case of the Cholesky factorization ($L = \Gamma(A)$), matrix A is symmetric, therefore we partition it into four quadrants

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array} \right),$$

where A_{TL} and A_{BR} are square submatrices, so that they inherit the symmetric structure; here $Size(A_{TL}) = (k_1 \times k_1)$, with $0 \leq k_1 \leq m(L)$. The partitioned matrix A is substituted into the postcondition ($LL^T = A$), inducing a partitioning of the triangular matrix L:

$$L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right)$$

with $Size(L_{TL}) = (k_1 \times k_1)$; notice that the quadrants L_{TL} and L_{BR} inherit the triangular structure. The postcondition becomes

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c|c} L_{TL}^T & L_{BL}^T \\ \hline 0 & L_{BR}^T \end{array}\right) = \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

and multiplying out

$$\left(\begin{array}{c|c} L_{TL}L_{TL}^T = A_{TL} & * \\ \hline L_{BL}L_{TL}^T = A_{BL} & L_{BL}L_{BL}^T + L_{BR}L_{BR}^T = A_{BR} \end{array}\right)$$

where the $*$ indicates that the matrix is symmetric. Equation $L_{TL}L_{TL}^T = A_{TL}$ corresponds to the Cholesky factorization of A_{TL}, i.e., $L_{TL} = \Gamma(A_{TL})$. Similarly $L_{BL}L_{BL}^T + L_{BR}L_{BR}^T = A_{BR}$ corresponds to $L_{BR} = \Gamma(A_{BR} - L_{BL}L_{BL}^T)$, while equation $L_{BL}L_{TL}^T = A_{BL}$ is a triangular linear system. These considerations lead to the expression

$$\left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL}L_{TL}^{-T} & L_{BR} = \Gamma(A_{BR} - L_{BL}L_{BL}^T) \end{array}\right), \tag{2.6}$$

which is the PME for the Cholesky factorization (in order to simplify the discussion, here we assume that matrix L does not overwrite A). One comment on the partitioning: if L is the Cholesky factor of A, then the equalities in (2.6) hold *true* independently of the size of quadrant A_{TL}, as long as it is square, including the boundary case $Size(L_{TL}) = (0 \times 0)$. This means that the PME is a recursive definition with no constraints on the size of the subproblems (see Section 2.6).

In order to determine possible loop-invariants from (2.6), we begin by studying data dependencies. Quantity L_{TL} is only a function of A_{TL}, therefore can be directly computed. Quadrant L_{BL} depends on A_{BL} as well as L_{TL}, so it should be computed only once L_{TL} is available. Likewise, L_{BR} is a function of A_{BR} and L_{BL}. This leads to a dependency chain: $L_{TL} \rightarrow L_{BL} \rightarrow L_{BR}$, meaning that one quantity

in the chain should be computed only once all the preceding quantities have been computed.

Let us now look at the operations appearing in (2.6). We derive loop invariants from the PME by deciding, for each operation, whether it is performed or not, while at the same time satisfying the dependency chain. The TL quadrant presents only one operation, $L_{TL} = \Gamma(A_{TL})$, which corresponds to the first entry in the dependency chain. Thus, in any loop-invariant, this operation needs to be performed (otherwise no other operation can be performed). Quadrant BL also presents one operation only, the triangular system $L_{BL}L_{TL}^T = A_{BL}$ (L_{BL} is the unknown), leading to two possible loop-invariants: one in which L_{BL} is left untouched $\left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & 0 \\ \hline L_{BL} = 0 & - \end{array}\right)$, and another where the system is solved $\left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL}L_{TL}^{-T} & - \end{array}\right)$ (the BR quadrant shows a dash because we have not discussed it yet). Finally, the BR quadrant contains an assignment which consists of more than a single operation: in order to compute $L_{BR} = \Gamma(A_{BR} - L_{BL}L_{BL}^T)$, the update $L_{BR} = A_{BR} - L_{BL}L_{BL}^T$ has to be performed first. In terms of loop-invariants, keeping in mind that L_{BR} makes use of L_{BL}, we have the choice of a) not performing any operation on L_{BR}, b) performing the partial update $L_{BR} = A_{BR} - L_{BL}L_{BL}^T$, or c) executing the full assignment $L_{BR} = \Gamma(A_{BR} - L_{BL}L_{BL}^T)$:

$$
\begin{array}{ll}
\text{a)} & \left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = 0 & L_{BR} = 0 \end{array}\right) \\
\text{b)} & \left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL}L_{TL}^{-T} & L_{BR} = 0 \end{array}\right) \\
\text{c)} & \left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL}L_{TL}^{-T} & L_{BR} = A_{BR} - L_{BL}L_{BL}^T \end{array}\right) \\
\text{d)} & \left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL}L_{TL}^{-T} & L_{BR} = \Gamma(A_{BR} - L_{BL}L_{BL}^T) \end{array}\right).
\end{array}
$$

#	Loop − invariant
1	$\left(\begin{array}{c\|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = 0 & L_{BR} = 0 \end{array}\right)$
2	$\left(\begin{array}{c\|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL}L_{TL}^{-T} & L_{BR} = 0 \end{array}\right)$
3	$\left(\begin{array}{c\|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL}L_{TL}^{-T} & L_{BR} = A_{BR} - L_{BL}L_{BL}^{T} \end{array}\right)$

Table 2.2: Loop-invariants for computing the Cholesky factor L of a symmetric positive definite matrix A.

The last loop-invariant coincides with the PME and therefore is not feasible: it corresponds to the state in which the solution has been fully computed, which is not a state that can be maintained *true* at each step of the computation. Table 2.2 summarizes the three feasible loop-invariants for computing the Cholesky factorization obtained by selecting sub-expressions of the PME.

2.5.2 Feasible Loop-Invariants

Different loop-invariants are derived from the PME by considering individual operations that contribute to the final result. Each such operation may or may not have been performed at an intermediate stage: a loop-invariant is a subset of the operations appearing in the PME. On the other hand, not every subset of the operations from the PME (or simply a 'subset' from now on) yields a feasible loop-invariant. We have already seen that the PME carries information about dependencies among quadrants: one condition for a subset to be feasible is that it must satisfy the de-

pendency chains. As an example, the subset

$$\left(\begin{array}{c|c} L_{TL} = 0 & * \\ \hline L_{BL} = A_{BL}L_{TL}^{-T} & L_{BR} = 0 \end{array}\right)$$

is not a loop-invariant for the Cholesky factorization because the BL quadrant contains an expression that refers to the quantity L_{TL} which has not been computed yet ($L_{TL} = 0$). Generally, a subset may not be feasible for one of the following reasons:

1. No simple initialization exists such that $\{P_{\text{pre}}\}$ Init $\{P_{\text{inv}}\}$.

   This is the case for the subset marked as d) for the Cholesky factorization (in order to satisfy the triple, the statement $L_{TL} = \Gamma(A)$ should be part of the initialization):

   $$\left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL}L_{TL}^{-T} & L_{BR} = \Gamma(A_{BR} - L_{BL}L_{BL}^{T}) \end{array}\right).$$

2. No loop guard exists such that $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$.

   One example is given by the subset that computes nothing, the "empty subset:"

   $$\left(\begin{array}{c|c} L_{TL} = 0 & * \\ \hline L_{BL} = 0 & L_{BR} = 0 \end{array}\right).$$

3. Data dependency. A loop-invariant which does not satisfy the dependency chains would result in an algorithm performing extra computations and/or overwriting data that is subsequently needed.

4. Extra storage. The subexpression is not of the same size as the quadrants or overwrites data that is subsequently needed.

A theory providing necessary and sufficient conditions for a subset to be a feasible loop-invariant is a future research goal, not within the scope of this dissertation.

2.5.3 Example: Triangular Discrete Time Sylvester Equation

The operation is defined as

$$\begin{aligned} P_{\text{pre}} \quad : \quad & \{\ \text{UpperTriangular}(A)\ \wedge\ Size(A) = (m \times m)\ \wedge \\ & \quad \text{LowerTriangular}(B)\ \wedge\ Size(B) = (n \times n)\ \wedge \\ & \quad Size(C) = (m \times n)\ \wedge\ Size(X) = (m \times n)\ \wedge \\ & \quad \text{Output}(X)\ \wedge\ \ldots\ \}; \\ P_{\text{post}} \quad : \quad & \{\ AXB - X = C\ \}. \end{aligned}$$

The dots in the precondition indicate that more conditions should be listed to ensure the existence of a solution. We will use the notation $X = \Psi(A, B, C)$ to denote that the matrix X is a solution for the triangular discrete time Sylvester (DTSY) equation $AXB - X = C$. We want to determine the PME for this operation.

Matrix A is triangular, so we partition it into four quadrants

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right)$$

where A_{TL} and A_{BR} are square submatrices, so that they possess the upper triangular structure: $Size(A_{TL})$ is $(k_1 \times k_1)$, with $0 \leq k_1 \leq m$.

When the partitioned matrix A is substituted into the postcondition, the partitionings of the other variables are induced. In order for the product AX (from the postcondition) to be well defined, matrix X has to be partitioned vertically into

two submatrices or into four quadrants

$$X \rightarrow \left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right) \quad \text{or} \quad X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array} \right),$$

with $m(X_T) = k_1$, $Size(X_{TL}) = (k_1 \times k_2)$ and $0 \leq k_2 \leq n$. Matrices B and C are then partitioned accordingly. It should not come as a surprise the fact that there may exist more than one PME for a target operation: depending on how the operands are partitioned, more than one expression can be derived. In our example, if we partition X vertically, then matrix B has to remain unpartitioned and matrix C has to be partitioned vertically, whereas if X is partitioned into quadrants, then matrices B and C have to be partitioned into four quadrants too. Let us examine the first scenario

$$B \rightarrow B \quad \text{and} \quad C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array} \right),$$

with $m(C_T) = k_1$; it follows that matrices C and X are partitioned the same way. Substituting the repartitioned matrices into the postcondition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array} \right) \left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right) B - \left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right) = \left(\begin{array}{c} C_T \\ \hline C_B \end{array} \right),$$

and multiplying the operands out and adding together gives

$$\left(\begin{array}{c} A_{TL}X_TB + A_{TR}X_BB - X_T = C_T \\ \hline A_{BR}X_BB - X_B = C_B \end{array} \right). \tag{2.7}$$

This expression indicates how the submatrices X_T and X_B of the solution matrix are related to the submatrices of the input variables. Equation 2.7 can be rewritten in

#	Loop − invariant
1	$\left(\begin{array}{c} X_T = 0 \\ \hline X_B = \Psi(A_{BR}, B, C_B) \end{array}\right)$
2	$\left(\begin{array}{c} X_T = C_T - A_{TR}X_B B \\ \hline X_B = \Psi(A_{BR}, B, C_B) \end{array}\right)$

Table 2.3: Loop-invariants for computing the solution of the triangular discrete time Sylvester equation.

terms of Ψ noticing that the matrices A_{BR} and A_{TL} are upper triangular, obtaining

$$\left(\begin{array}{c} X_T = \Psi(A_{TL}, B, C_T - A_{TR}X_B B) \\ \hline X_B = \Psi(A_{BR}, B, C_B) \end{array}\right), \tag{2.8}$$

which is one PME for the DTSY equation. Eqn. (2.8) indicates that given a vertical partitioning of the solution matrix X, the bottom part X_B, regardless of its size, always contains the solution of another triangular Sylverster equation, and the same for X_T. Furthermore, the PME reveals that X_B is defined only in terms of input variables (A_{BR}, B and C_B), while X_T is a function of input variables (A_{TL}, A_{TR}, B, C_T) as well as X_B. Therefore X_T should be computed after X_B: $X_B \rightarrow X_T$.

An analysis of the operations appearing in the PME, together with the dependency chain, results in two loop-invariants, as displayed in Table 2.3.

Recall that in the derivation of the PME we could have made a different choice regarding the partitioning of matrix X. Let us now examine the case in which matrix X is partitioned into four quadrants: this forces matrices B and C to be partitioned into quadrants too

$$X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right), \quad B \rightarrow \left(\begin{array}{c|c} B_{TL} & 0 \\ \hline B_{BL} & B_{BR} \end{array}\right) \quad \text{and} \quad C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right),$$

where $Size(X_{TL}) = (k_1 \times k_2)$, $Size(B_{TL}) = (k_2 \times k_2)$ and $Size(C_{TL}) = (k_1 \times k_2)$. The quadrants A_{TL}, A_{BR}, B_{TL} and B_{BR} are triangular matrices. Substituting the repartitioned matrices into the postcondition:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right)\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)\left(\begin{array}{c|c} B_{TL} & 0 \\ \hline B_{BL} & B_{BR} \end{array}\right) - \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) = \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right),$$

and multiplying the operands out and adding together

$$\left(\begin{array}{c|c} \begin{array}{r} (A_{TL}X_{TL} + A_{TR}X_{BL})B_{TL} + \\ (A_{TL}X_{TR} + A_{TR}X_{BR})B_{BL} - \\ X_{TL} = C_{TL} \end{array} & \begin{array}{r} (A_{TL}X_{TR} + A_{TR}X_{BR})B_{BR} - \\ X_{TR} = C_{TR} \end{array} \\ \hline \begin{array}{r} A_{BR}(X_{BL}B_{TL} + X_{BR}B_{BL}) - \\ X_{BL} = C_{BL} \end{array} & A_{BR}X_{BR}B_{BR} - X_{BR} = C_{BR} \end{array}\right);$$

since the diagonal quadrants of A and B are triangular, we can rewrite the expression in terms of Ψ functions

$$\left(\begin{array}{c|c} \begin{array}{l} X_{TL} = \\ \Psi(A_{TL}, B_{TL}, \\ \quad C_{TL} - A_{TR}X_{BL}B_{TL} - \\ \quad A_{TL}X_{TR}B_{BL} - A_{TR}X_{BR}B_{BL}) \end{array} & \begin{array}{l} X_{TR} = \\ \Psi(A_{TL}, B_{BR}, \\ \quad C_{TR} - A_{TR}X_{BR}B_{BR}) \end{array} \\ \hline \begin{array}{l} X_{BL} = \\ \Psi(A_{BR}, B_{TL}, C_{BL} - A_{BR}X_{BR}B_{BL}) \end{array} & X_{BR} = \Psi(A_{BR}, B_{BR}, C_{BR}) \end{array}\right) \quad (2.9)$$

which is another PME for the DTSY equation. It illustrates that the solution X of equation $AXB - X = C$ has the following property: when X is divided into four regions, each region, independently of the size, contains the solution of a DTSY equation defined by quadrants of matrices A, B, C and X. Equation (2.9) not only expresses how the four different quadrants can be computed, but it also reveals the dependencies among the quadrants of X: X_{BR} can be computed directly in

terms of input variables (A_{BR}, B_{BR} and C_{BR}), both X_{BL} and X_{TR} depend on X_{BR} and quadrant X_{TL} depends on X_{TR}, X_{BL} and X_{BR}. This can be summarized by the two dependency chains $X_{BR} \rightarrow X_{BL} \rightarrow X_{TL}$ and $X_{BR} \rightarrow X_{TR} \rightarrow X_{TL}$. As a consequence, the computation of X has to begin from the BR corner ($X_{BR} = \Psi(A_{BR}, B_{BR}, C_{BR})$) and proceeds by expanding the BR quadrant towards the TL corner.

By looking at (2.9) it is not hard to guess that this operation is exceptionally rich in loop-invariants: quadrants BL, TR and TL need to be updated before the corresponding Sylvester equation can be solved (quadrant TL in particular needs to be updated with the contributions by the other three quadrants). This means that different loop-invariants are generated depending on whether each quadrant a) is not updated, b) is partially computed (updated to some extent), or c) is fully computed. A careful enumeration that takes dependencies into account unveils 32 different variants.

For completeness we mention that yet another partitioning of matrices A, B, C and X is feasible, resulting in a third PME for the DTSY equation:

$$X \rightarrow \left(\begin{array}{c|c} X_L & X_R \end{array} \right), \quad B \rightarrow \left(\begin{array}{c|c} B_{TL} & 0 \\ \hline B_{BL} & B_{BR} \end{array} \right) \quad \text{and} \quad C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array} \right),$$

with A left unpartitioned. The PME is

$$\left(\begin{array}{c|c} X_L = \Psi(A, B_{TL}, C_L - AX_RB_{BL}) & X_R = \Psi(A, B_{BR}, C_R) \end{array} \right), \tag{2.10}$$

from which two feasible loop-invariants can be identified.

2.6 Loop-based and Recursive Algorithms

The worksheet in Fig. 2.2 is a skeleton for loop-based algorithms: it consists of an initialization followed by a while loop. It is easy to jump to the wrong conclusion that our methodology does not cover recursive algorithms. In reality, a recursive algorithm is equivalent to a blocked algorithm in which b —the block size— is chosen so that the problem is split in halves.[2] With reference to Fig 2.2, depending on the choice of parameter b in the **Repartition** statement (Step 5a), the worksheet generates algorithms that are unblocked ($b = 1$), blocked ($b > 1$) or recursive ($b = m(X)/2$ or $b = n(X)/2$, where X is one of the operands).

Despite the iterative nature of our methodology, recursion plays an important role: it is typical that in computing an operation $\mathcal{O}p$, the same $\mathcal{O}p$ is performed with smaller operands. The algorithm for computing the inverse of a triangular matrix that we derived in Fig. 2.3 serves as an example: at each iteration, the triangular submatrix L_{11} has to be inverted. The computation of the Cholesky factor in Fig 1.4 (right) exhibits the same observation: one of the updates requires the computation of the Cholesky factorization of submatrix A_{11}. For both examples, the subproblem may be solved in different ways according to its size (submatrix L_{11} for the triangular inversion, A_{11} for the Cholesky factorization): when L_{11} and A_{11} are 1×1 matrices, the subproblems reduce to scalar operations; if instead the matrix size (> 1) is such that the problem fits in cache, then unblocked algorithms are often deployed; finally, if the problem is size is so large that does not fit in cache, then blocked algorithms are used. The latter two cases involve a recursive call.

In general, it is the PME that dictates whether one algorithm has recursive calls or not: the PME illustrates how one operation $\mathcal{O}p$ can be solved by assembling the solutions of subproblems. Often times these subproblems require the computa-

[2] We are intentionally ignoring tail-recursive algorithms, as they are easily expressed as iterative algorithms.

tion of $\mathcal{O}p$ itself, hence the recursive calls. Since the PME is a recursive definition of the target operation, devising recursive algorithms is straightforward (the converse is also true). The rationale for loop-based algorithms is high-performance.

A loop-based approach allows the users to carefully select the size of the submatrices exposed by the **Repartition** statement. Since it is these submatrices that are involved in the computational statements, selecting their size is crucial to attain high-performance: a block size too small prevents the cost for data movement to be amortized by computations (the BLAS does not reach peak performance), while a block size that is too big causes excessive data movement (the cache memory is not big enough to contain the operands). The optimal block size depends on the operation performed and the target architecture. A purely recursive approach does not naturally permit such block size fine-tuning.

A question is whether our methodology covers every recursive algorithm: given an operation $\mathcal{O}p$ and a known recursive algorithm $\mathcal{A}_R$ computing $\mathcal{O}p$, can an iterative version of $\mathcal{A}_R$ be obtained by filling in the worksheet? The answer is yes, if the recursive algorithm does not require fixed sizes for the subproblems: in this case, algorithm $\mathcal{A}_R$ corresponds to a PME (see Section 2.5), therefore the methodology can be applied. One example of recursive algorithm with no size constraints on the subproblems is Mergesort: the output sorted array results from merging two sorted sub-arrays, independently of their size. Therefore it is possible to derive a loop-based Mergesort. By contrast, Strassen's algorithm for multiplying two matrices requires that at each step the matrices are partitioned into four quadrants of equal size. This prevents the operands from being traversed incrementally; consequently our methodology cannot be applied.

2.7 Summary

We proposed a framework for systematic derivation of programs that exploits the typical structure of loop-based linear algebra algorithms. The framework itself exploits the Floyd-Hoare logic for proving correctness of programs. Central to this logic are the concepts of Hoare triple and loop-invariant. This chapter expands upon our paper published in the ACM Transactions on Mathematical Software [5] which itself was a refinement of an earlier paper by Gunnels et al. [24]. It places the derivation process within the rigid setting of formal methods, as advocated by pioneers like E.W. Dijkstra and C.A.R. Hoare. The result is a template of a proof of correctness that serves as a worksheet for deriving algorithms.

Highlights to take away from this chapter follow:

- The procedure for deriving algorithms is fully determined once a loop-invariant is identified. The derivation is therefore systematic.
- The PME is the predicate that allows us to systematically devise —*a priori*— loop-invariants for algorithms for the target operation.
- Since typically many loop-invariants can be identified from the PME, our methodology yields many algorithms for a given target operation.

This chapter contributes to the field of formal derivation by demonstrating that, for a class of linear algebra operations, it is possible to systematically build correct loop-based algorithms starting solely from the mathematical definition of the operations. The proof of correctness guides the algorithm construction, instead of following it.

With respect to dense linear algebra libraries, we provided evidence that the systematic derivation of families of correct algorithms is achievable. This is a first step towards a mechanical system for deriving libraries.

Chapter 3

Mechanical Derivation of Algorithms

In Chapter 2 we have demonstrated how formal derivation techniques can be applied to linear algebra to derive formally correct families of loop-based algorithms. The block variants of these algorithms can attain high-performance by virtue of the fact that most computation is cast in terms of matrix-matrix operations, like the ones included in the level 3 BLAS library. Our next goal is to mechanically generate linear algebra algorithms starting from the mathematical specification of the target operation only. Ultimately, one should be able to visit a website, fill in a form with information about the operation to be performed, choose a programming language, click the SUBMIT button, and receive a library of routines that compute the operation. This chapter provides evidence that for a large number of operations this vision in within reach.

While the derivation procedure discussed in Chapter 2 ensures the derived algorithm is correct, it is the application of the methodology itself that is error prone when performed by a human. It involves tedious algebraic manipulations. As the complexity of the target operation increases, it becomes more cumbersome to fill in

the worksheet by hand. Furthermore, in order to generate families of algorithms, the worksheet has to be completed as many times as the number of loop-invariants that the PME admits.

It is important to realize that the procedure itself does not introduce unnecessary elements of confusion. Operations once deemed "for experts only" can now be tackled by undergraduates, leaving more ambitious problems for the experts. A concrete example is given by the operation that computes the solution to the triangular Sylvester equation $AX + XB = C$. A few algorithms for solving the Sylvester equation have been known for about 30 years [2]. Nonetheless, new variants are still discovered and published with some regularity [31, 29]. Our derivation methodology has been applied to this operation yielding a family of 16 algorithms, including the algorithms that were already known as well as many undiscovered ones [34]. The complication lies in that, as part of the derivation, complex matrix expressions are introduced that require simplification, providing an opportunity for algebra mistakes to be made by a human. Indeed, one of the variants derived in [34] did not return the correct outcome when implemented and executed. Mistakes in the simplifications involved in Step 8 (determining the updates) were only pinpointed when the updates were re-derived with the aid of the mechanical system that we describe in this chapter (Section 3.2).

Before diving into the details of our mechanical system, we want to clarify again the difference between the concepts of automatic tuning and mechanical derivation of algorithms. Tuning is a process that begins with a parameterized routine and attempts to automatically determine optimal parameters [11, 38, 42]. Our mechanical derivation of algorithms is a process that takes the mathematical specification of a target operation as input (instead of an algorithm), and returns a family of formally correct algorithms for the input operation. There is early evidence that indicates that our derivation methodology yields a larger family of algorithms than

those obtained by applying compiler transformations to an individual member of the family. This topic is still an active research question beyond the scope of this dissertation.

3.1 Towards a Mechanical Procedure

Once a loop-invariant has been selected (P_{inv} in Step 2), all the following steps in the procedure presented in Sec. 2.3 are completely determined: they can be expressed as functions of the predicates P_{pre}, P_{post}, PME and P_{inv}. Here we revisit the eight steps with an eye on opportunities for mechanization. Throughout this section, the boxed text refers to the triangular discrete time Sylvester equation (DTSY, Sec. 2.5.3), which we use as example target operation. The derivation of algorithms for this operation is then completed in the next section, with the help of a mechanical system that we developed.

1 P_{pre} **and** P_{post} **and** PME. The precondition, the postcondition and the PME are given as part of the specifications for the operation that we want to implement. They are the input to a mechanical system; no computation is required.

For the DTSY equation, P_{pre} and P_{post} are:

$$
\begin{aligned}
P_{\text{pre}} \quad : \quad & \{\ \text{UpperTriangular}(A)\ \wedge\ \mathit{Size}(A) = (m \times m)\ \wedge \\
& \quad \text{LowerTriangular}(B)\ \wedge\ \mathit{Size}(B) = (n \times n)\ \wedge \\
& \quad \mathit{Size}(C) = (m \times n)\ \wedge\ \mathit{Size}(X) = (m \times n)\ \} \\
& \quad \text{Output}(X)\ \wedge\ \ldots\}, \\
P_{\text{post}} \quad : \quad & \{\ AXB - X = C\ \}.
\end{aligned}
$$

The three PMEs are given in Section 2.5.3.

In actuality, the PMEs are not arguments to be passed to a mechanical system as input. Instead, they constitute a library of definitions, in the form of rewrite rules that specify how the solution to one problem can be computed by solving simpler sub-problems. A mechanical system should have the capability of deducing this information for the basic operations, like matrix-matrix multiplications and additions. For more complex operations, a database of PMEs can serve as knowledge to decide how to decompose the computation of an operation into simpler operations.

In general, the derivation of algorithms for an operation $\mathcal{O}p$ requires knowledge of more PMEs than just those for $\mathcal{O}p$. As an example, we consider a special case of the DTSY equation. Let us assume that B equals A^T, then the equation reduces to $AXA^T - X = C$, that is the triangular discrete time Lyapunov equation. The derivation of algorithms for this equation involves its PME as well as the PMEs for the DTSY equation. The list of PMEs needed to derive algorithms for $\mathcal{O}p$ is determined by looking at $\mathcal{O}p$'s PMEs.

We present here the PMEs, in the form of rewrite rules, for the DTSY equation. Recall that the notation $X = \Psi(A, B, C)$ denotes that matrix X is a solution for the equation $AXB - X = C$. In Section 2.5.3 we had established that if A is partitioned into four quadrants, C and X are partitioned vertically and B is left unpartitioned, then (Eqn. (2.8))

$$\left(\begin{array}{l} X_T = \Psi(A_{TL}, B, C_T - A_{TR} X_B B) \\ \hline X_B = \Psi(A_{BR}, B, C_B) \end{array} \right),$$

which is a PME for the DTSY equation. This PME can also be written as

$$\left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right) = \Psi \left(\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array} \right), B, \left(\begin{array}{c} C_T \\ \hline C_B \end{array} \right) \right) = \left(\begin{array}{l} \Psi(A_{TL}, B, C_T - A_{TR} X_B B) \\ \hline \Psi(A_{BR}, B, C_B) \end{array} \right), \quad (3.1)$$

which represents a rewrite rule for the function Ψ: every time Ψ is invoked with arguments having the structure described in Eqn. (3.1), it can be replaced by the right hand side of Eqn. (3.1).

Similarly, the second and third PMEs (Eqns. (2.9) and (2.10)) can respectively be written as

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) = \Psi\left(\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right), \left(\begin{array}{c|c} B_{TL} & 0 \\ \hline B_{BL} & B_{BR} \end{array}\right), \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)\right) = \tag{3.2}$$

$$\left(\begin{array}{l|l} \Psi(A_{TL}, B_{TL}, & \Psi(A_{TL}, B_{BR}, \\ \quad C_{TL} - A_{TR}X_{BL}B_{TL} - & \quad C_{TR} - A_{TR}X_{BR}B_{BR}) \\ \quad A_{TL}X_{TR}B_{BL} - A_{TR}X_{BR}B_{BL}) & \\ \hline \Psi(A_{BR}, B_{TL}, & \\ \quad C_{BL} - A_{BR}X_{BR}B_{BL}) & \Psi(A_{BR}, B_{BR}, C_{BR}) \end{array}\right)$$

and

$$\left(\begin{array}{c|c} X_L & X_R \end{array}\right) = \Psi\left(A, \left(\begin{array}{c|c} B_{TL} & 0 \\ \hline B_{BL} & B_{BR} \end{array}\right), \left(\begin{array}{c|c} C_L & C_R \end{array}\right)\right) = \tag{3.3}$$

$$\left(\begin{array}{c|c} \Psi(A, B_{TL}, C_L - AX_RB_{BL}) & \Psi(A, B_{BR}, C_R) \end{array}\right).$$

Eqns. (3.3), (3.3) and (3.3) are rewrite rules for function Ψ, when invoked with arguments A, B and C partitioned as indicated above. These rewrite rules are necessary definitions needed by a mechanical system.

2 **Determine P_{inv}.** Loop-invariants are identified by selecting a subset of the operations that appear in the PME. In Section 2.5.2 we pointed out that not every subset leads to a viable algorithm: dependencies and the dimensions of the sub-expressions need to be taken into account. A mechanical system can build the dependency chain and keep track of the dimensions for each PME subexpression, thus discarding infeasible choices of loop-invariants.

In our example we will focus on the the following loop-invariant

$$P_{\text{inv}} : \left(\begin{array}{c|c} X_{TL} = \hat{C}_{TL} & X_{TR} = \Psi(A_{TL}, B_{BR}, \hat{C}_{TR} - A_{TR}X_{BR}B_{BR}) \\ \hline X_{BL} = \hat{C}_{BL} & X_{BR} = \Psi(A_{BR}, B_{BR}, \hat{C}_{BR}) \end{array}\right), \tag{3.4}$$

corresponding to an algorithm that computes the solution matrix X column-wise, from left to right, overwriting matrix C: notice that both the BR and TR quadrants contain the corresponding expression in the PME, while the BL and TL are left untouched (they contain the initial values $\hat{C}_{BL}$ and $\hat{C}_{TL}$, respectively).

3 **Determine the initialization.** Since the derivation methodology requires no computation to be performed at this stage, the initialization reduces to partitioning some or all the variables (from P_{pre}) in such a way that, for each variable that is partitioned, at least one of the submatrices (vectors are a special case of matrices) is null.[1] A mechanical system can exhaustively try out all the possible partitionings (conformally to the matrix properties describes in P_{pre}), selecting the ones that render *true* the implication $P_{\text{pre}} \implies P_{\text{inv}}$. If no such partitioning is found, the loop-invariant is labelled as infeasible and no further steps are executed.

The loop-invariant that we selected is originated from the second PME (Eqn. (2.9)), which comes about when the operands A, B, C and X are all partitioned into quadrants. Once the dimension for the partitioning of X is specified ($X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$, with $Size(X_{xy}) = 0 \times 0$, for a choice of $xy \in$ [TL, TR, BR, BL]), the dimensions for the partitioning of all the other matrices are determined. Out of the four alternatives, only the partitioning

$$X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \quad \text{where} \quad X_{BR} \ \text{is} \ 0 \times 0$$

is such that the precondition implies P_{inv}.

[1] A matrix is null if one or both its dimensions are zero.

By virtue of conformal partitioning, the corresponding initialization statements for the other variables are

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right),\ B \rightarrow \left(\begin{array}{c|c} B_{TL} & 0 \\ \hline B_{BL} & B_{BR} \end{array}\right),\ C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$$

where $A_{BR},\ B_{BR},\ C_{BR}$ are 0×0

4 **Determine loop-guard G.** The variables that have been partitioned as part of the initialization contain at least one null submatrix. At each iteration of the loop, one of the empty submatrices is expanded, until it encompasses the entire matrix. The loop-guard is given by a predicate of the form "the null submatrix Z_{xy} is not equal in size to the entire matrix Z;" the submatrix xy can be found by a system, testing exhaustively all the alternatives and selecting one for which the implication $P_{\text{inv}} \wedge \neg G \implies P_{\text{post}}$ is *true*. If no such partitioning is found, the loop-invariant is labelled as infeasible and no further steps are executed.

We determined that partitioning X into four quadrants with X_{BR} being 0×0 is one of the initialization statements. Quadrant X_{BR} grows at each iteration, therefore the predicate $Size(X_{BR}) \neq Size(X)$ can be used as loop-guard.

5 **Determine how to move boundaries.** How to traverse through the variables follows directly from the initialization and the loop-guard. The **Repartition** and **Continue** statements are responsible to make progress towards making G *false* by increase the size of the initially null matrix in G. Every other variable is then re-partitioned conformally. Furthermore, operands with a particular structure (triangular, symmetric, diagonal) can only be partitioned and traversed in a way that preserves the structure. This analysis can be made mechanically.

Matrix X is traversed by first exposing new regions (**Repartition**) from the TL, TR and BL quadrants, and then adding them to the BR quadrant (**Continue**). This is accomplished by the statements

Repartition

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} X_{00} & X_{01} & X_{02} \\ \hline X_{10} & X_{11} & X_{12} \\ \hline X_{20} & X_{21} & X_{22} \end{array}\right)$$

and

Continue with

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} X_{00} & X_{01} & X_{02} \\ \hline X_{10} & X_{11} & X_{12} \\ \hline X_{20} & X_{21} & X_{22} \end{array}\right).$$

Similar statements are derived for matrices A, B and C.

6 **Determine** P_{before}. This predicate is the loop-invariant expressed in terms of newly exposed parts of the operands, right after the **Repartition** statements. The expression for P_{before} is computed by 1) applying the textual substitution rules dictated by **Repartition** to the P_{inv}, 2) expanding by means of the rewrite rules defined by the PMEs and linear algebra and, 3) simplifying. Performing the textual substitution is straightforward, while the expansion and simplification of the expressions requires powerful symbolic computation tools.

The last three steps for the DTSY equation are performed with the aid of a mechanical system that is the topic of the next section.

7 **Determine** P_{after}**.** The computation of this predicate in analogous to the computation of P_{before} except for that the textual substitution rules are dictated by the **Continue** statements.

8 **Determine the update** S_U**.** The updates are determined by a comparison of the states P_{before} and P_{after}. This step involves a great deal of pattern matching. It can be made mechanical by using a symbolic system with pattern matching capabilities.

3.2 A Mechanical System

We present here a prototype mechanical system that requires limited human intervention. We describe in details the steps that a user has to follow in order to use the system for the generation of algorithms and routines. A list of examples follows.

3.2.1 Generating Algorithms

Not every step in the derivation procedure is complex. Step 1 requires no computation. The task of identifying one or more loop-invariants from the PME (Step 2) is relatively simple. Once P_{inv} is selected, Steps 3, 4 and 5 not only are trivial, but often times they are equal across many algorithms in the family. The bulk of the complexity lies with Steps 6, 7 and 8; these steps require symbolic computations and are different for each loop-invariant. The predicate P_{before} (Step 6) expresses the contents of the output operands at the top of each iteration (current state); P_{after} (Step 7) indicates what the output operands need to contain at the bottom of the loop, and S_U (Step 8) contains the computational statements to transition from P_{before} to P_{after}.

In order to facilitate the computation of P_{before}, P_{after} and S_U, we developed a prototype mechanical system. We started implementing an environment to perform symbolic operations with blocked matrices.[2] Then we wrote a system for manipulating matrix functions by means of substitution rules, simplifications and pattern matching. Finally we added a graphical output to display algorithms in a format that closely resembles the worksheet (Fig. 2.2). We chose Mathematica [41] as the programming environment and language because of its powerful symbolic computation and pattern matching capabilities.

In this section we illustrate the steps that a user has to execute in order to

[2] A matrix is blocked if the entries are also matrices (including the boundary cases of vectors, scalars and null matrices). Regardless of whether the target algorithm is blocked or unblocked, its derivation involves computations with blocked matrices.

generate algorithms for the DTSY equation. The input to the system is a loop-invariant and P_{pre}; the output is either the description of an algorithm or a routine computing the target operation (see Sec. 3.2.2). As we pointed out in the last section, the operation's PMEs are needed in the form of rewrite rules.

```
DTSY[
    {{aTL_, aTR_},
     {0,    aBR_}},

    {{bTL_, 0},
     {bBL_, bBR_}},

    {{cTL_, cTR_},
     {cBL_, cBR_}}
    ] :=

Module[{xBR, xTR, xBL, XTL},

  xBR = DTSY[aBR, bBR, cBR];
  xTR = DTSY[aTL, bBR, cTR - prod[aTR, xBR, bBR]];
  xBL = DTSY[aBR, bTL, cBL - prod[aBR, xBR, bBL]];
  xTL = DTSY[aTL, bTL, cTL - prod[aTR, xBL, bTL] -
                             prod[aTL, xTR, bBL] -
                             prod[aTR, xBR, bBL]];

  {{xTL, xTR},
   {xBL, xBR}}
]
```

Figure 3.1: Mathematica definition for PME (3.2).

We start by discussing how to define PMEs as rewrite rules. In Fig. 3.1 we show a definition, in Mathematica, corresponding to the PME (2.9). Notice the close similarity with (3.2): the top part of the definition (before the ":=") specifies the name of the function (DTSY, corresponding to Ψ) and the structure of the arguments that has to be satisfied for the right hand side to be evaluated. In this case the definition of DTSY applies only when the following three conditions are met simultaneously: 1) the first argument is a partitioned 2×2 upper triangular matrix

(the BL quadrant is explicitly 0), 2) the second argument is a partitioned 2×2 lower triangular matrix, 3) the third argument is a partitioned 2×2 matrix. The body of the function (after the ":=") consists of a preparative sequence of assignments to variables xBR, xTR, xBL, xTL (corresponding to quadrants $X_{BR}, X_{TR}, X_{BL}, X_{TL}$, respectively), followed by the return value. The assignments mirror the expressions in (2.9): `xBR = DTSY[aBR, bBR, cBR]`, for instance, is the translation for $X_{BR} = \Psi(A_{BR}, B_{BR}, C_{BR})$. Finally, the return value is assembled by the construct

`{{xTL, xTR},`
`{xBL, xBR}}` which corresponds to the 2×2 result matrix $\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$.

Similar definitions are specified for the other two PMEs (Eqns. (2.8) and (2.10)).

```
DTSY1[
     {{aTL_, aTR_},
      {0,    aBR_}},

     {{bTL_, 0},
      {bBL_, bBR_}},

     {{cTL_, cTR_},
      {cBL_, cBR_}}
     ] :=

Module[{xBR, xTR},

    xBR = DTSY[aBR, bBR, cBR];
    xTR = DTSY[aTL, bBR, cTR - prod[aTR, xBR, bBR]];

    {{cTL, xTR},
     {cBL, xBR}}
]
```

Figure 3.2: Mathematica definition for loop-invariant (3.4).

The definition for `DTSY1`, corresponding to the loop-invariant (3.4), is similarly given: Fig. 3.2. This definition is created by deleting parts from Fig. 3.1, much like a loop-invariant is created by selecting a subset of a PME. The function `DTSY1`

is the first argument to the mechanical system.

The PMEs have been encoded as rewrite rules and the loop-invariant has been defined; we are just one step away from being able to invoke the mechanical system: we only have to pass the information contained in P_{pre} as input. The second argument to the system is a list that describes the properties of each variable in P_{pre}:[3]

```
Pre = {{"A", "UpperTriangular", "TL"},
       {"B", "LowerTriangular", "TL"},
       {"C", "TL", "Overwrite"}};
```

The mechanical system can now be invoked via the call

```
worksheet[DTSY1, Pre];
```

and the output is a worksheet completed like the one in Fig. 2.3. Unfortunately, in this particular example, the expressions for P_{before}, and especially for P_{after}, are so long that they would not fit this page even when typeset with the smallest readable font; this is exactly why a mechanical system is needed! Fortunately, the resulting update statements (Step 8) are manageable. In Fig. 3.3 we display the blocked algorithm, derived by the mechanical system, corresponding to loop-invariant (3.4). The function DTSY is invoked in the updates: this could be a recursive call to the blocked algorithm or a call to an unblocked algorithm. Unblocked algorithms are also derived with our methodology, setting the block size to 1.

In the next section we present routines, mechanically generated, implementing the algorithm in Fig. 3.3. Examples and screen-shots of algorithms mechanically derived are given in Section 3.2.3.

[3]Strictly speaking, the second argument also contains a flag (`"TL"` in the DTSY example) about the initial partitioning of each operand, which is a piece of information not present in the precondition.

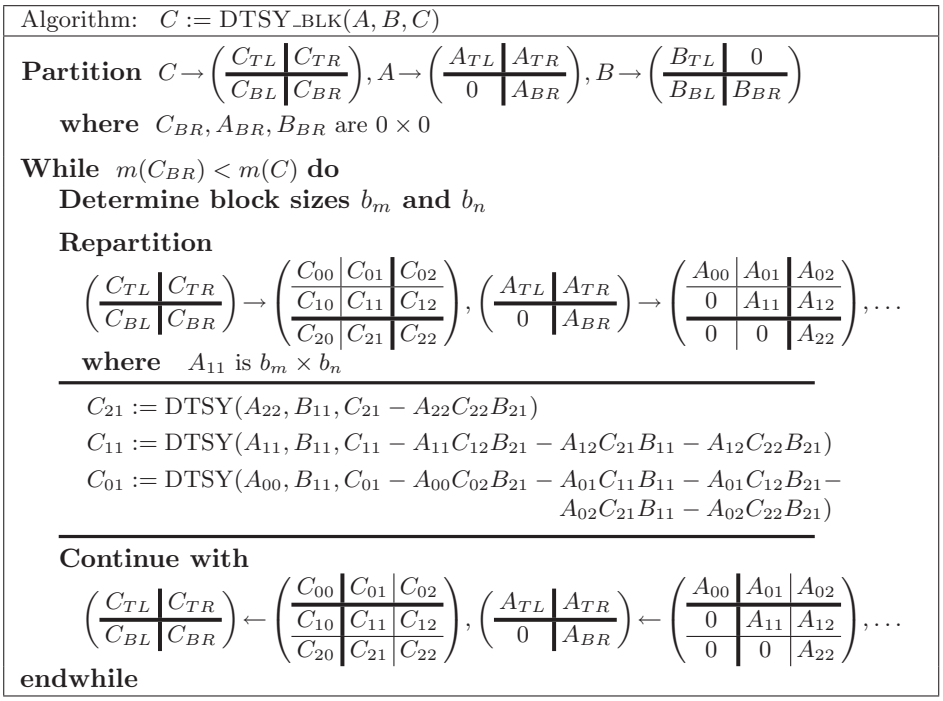

Figure 3.3: Mechanically derived blocked algorithm for computing the solution of the DTSY equation: loop-invariant (3.4).

3.2.2 Generating Code

Every algorithm derived with the FLAME methodology contains a combination of instructions to partition and repartition variables, one **While** loop, and computational statements. The FLAME APIs (Sec. A.2) [8] provide the users with the **Part**, **Repartition**, and **Continue** constructs for a number of different languages. It is possible to envision a modification of the mechanical system that returns routines for the derived algorithms. This goal can be accomplished by defining rewrite rules that translate the partitioning and repartitioning statements, the loop-skeleton and create the function's prototype into a target language. The remaining step, i.e., the translation of computational statements into a target language, is where the

challenge lies.

We extended our mechanical system to return Matlab ("FLAME@lab") or C ("FLAME/C") code. In Figs. 3.4 and 3.5 we display Matlab and C routines resulting from the Mathematica invocation of `worksheetMF[DTSY1, Pre]` and `worksheetCF[DTSY1, Pre]`, respectively (in the function call, `M` stands for Matlab, `C` indicates the C language, and the final `F` redirects the output to a file).[4]

The Matlab routine in Fig. 3.4 is executable once the function DTSY (invoked in the updates) is a known routine (one easy choice would be the unblocked version of the same algorithm). Scientific languages like Matlab and Mathematica are able to interpret and execute high level matrix expressions involving operations like multiplication, addition, transposition, inversion as well as function calls. Therefore the computational statements can be directly translated into these languages. On the contrary, the C routine in Fig. 3.5 requires editing: in order to attain high-performance, the updates should be written as a sequence of calls to the BLAS library. The translation of a matrix expression into a sequence of BLAS calls currently has to be performed by the user. Our system generates C routines annotated with the Matlab updates (as comments) to assist the user.

3.2.3 Examples

This section is a brief parade of mechanically generated algorithms and worksheets. In Fig. 3.3 we presented the algorithm for the DTSY equation, concluding the derivation process for loop-invariant (3.4). Let us now consider a different loop-invariant for the same operation:

$$\left(\begin{array}{c|c} X_{TL} = \hat{C}_{TL} & X_{TR} = \hat{C}_{TR} \\ \hline X_{BL} = \hat{C}_{BL} & X_{BR} = \Psi(A_{BR}, B_{BR}, \hat{C}_{BR}) \end{array}\right). \tag{3.5}$$

[4] The routines have been edited to nicely format the updates and to shorten the code: we included only the partitioning and repartitioning statements for matrix C.

```
function [C] = DTSY1( A , B , C , nb )
[..]
[ CTL, CTR, ...
  CBL, CBR ] = FLA_Part_2x2( C,0,0,'FLA_BR');

%% Loop Invariant
%% CTL=CTL
%% CTR=DTSY[ATL, BBR, CTR - ATR . DTSY[ABR, BBR, CBR] . BBR]
%% CBL=CBL
%% CBR=DTSY[ABR, BBR, CBR]

while( size(CBR,1) ~= size(C,1) | size(CBR,2) ~= size(C,2) )
   b = min( nb, min( size(CTL,1), size(CTL,2) ));
   [..]
   [ C00, C01, C02, ...
     C10, C11, C12, ...
     C20, C21, C22 ] = FLA_Repart_2x2_to_3x3(CTL, CTR,...
                                             CBL, CBR,...
                                             b, b, 'FLA_TL');
   %* ********************************************************** *%
   C21 = DTSY(A22, B11, C21 - A22 * C22 * B21);

   C11 = DTSY(A11, B11, C11 - A11 * C12 * B21 - A12*C21*B11 -
                                                A12*C22*B21 );

   C01 = DTSY(A00, B11, C01 - A00 * C02 * B21 - A01 * C11 * B11 -
                              A01 * C12 * B21 - A02 * C21 * B11 -
                                                A02 * C22 * B21 );
   %* ********************************************************** *%
   [..]
   [ CTL, CTR, ...
     CBL, CBR ] = FLA_Cont_with_3x3_to_2x2(C00, C01, C02, ...
                                           C10, C11, C12, ...
                                           C20, C21, C22, ...
                                           'FLA_BR');
end;
C = CBR;
return;
```

Figure 3.4: Mechanically generated FLAME@lab routine for computing the solution of the DTSY equation: loop-invariant (3.4).

```
void DTSY1( FLA_Obj A , FLA_Obj B , FLA_Obj C , int nb )
{ int  nb;
  [..]
  FLA_Obj  CTL, CTR,    C00, C01, C02,
           CBL, CBR,    C10, C11, C12,
                        C20, C21, C22;
  [..]
  FLA_Part_2x2( C,  &CTL, /**/ &CTR,
                  /* ************** */
                    &CBL, /**/ &CBR,
                0, /* by */ 0, /* submatrix */ FLA_BR );
%% Loop Invariant
%% CTL=CTL
%% CTR=DTSY[ATL, BBR, CTR - ATR . DTSY[ABR, BBR, CBR] . BBR]
%% CBL=CBL
%% CBR=DTSY[ABR, BBR, CBR]

  while( FLA_Obj_length(CBR) != FLA_Obj_length(C) )
  {
    b = min( nb, FLA_Obj_length(CTL) );
    [..]
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,       &C00, &C01, /**/ &C02
                                /**/            &C10, &C11, /**/ &C12,
                         /* ************ */   /* ******************** */
                           CBL, /**/ CBR,       &C20, &C21, /**/ &C22,
                  /*with*/ b, /*by*/ b, /* C11 split from */ FLA_TL);
%* ********************************************************************* *%
/* C21 = DTSY(A22, B11, C21 - A22 * C22 * B21); */

/* C11 = DTSY(A11, B11, C11 - A11 * C12 * B21 - A12 * C21 * B11 -
                                                A12 * C22 * B21 ); */

/* C01 = DTSY(A00, B11, C01 - A00 * C02 * B21 - A01 * C11 * B11 -
                              A01 * C12 * B21 - A02 * C21 * B11 -
                                                A02 * C22 * B21 ); */
%* ********************************************************************* *%
    [..]
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,      C00, /**/ C01, C02,
                            /* ************** */ /* ****************** */
                                    /**/            C10, /**/ C11, C12,
                              &CBL, /**/ &CBR,      C20, /**/ C21, C22,
                              /*with C11 added to submatrix */ FLA_TR);
   }
}
```

Figure 3.5: Mechanically generated C routine for computing the solution of the DTSY equation: loop-invariant (3.4).

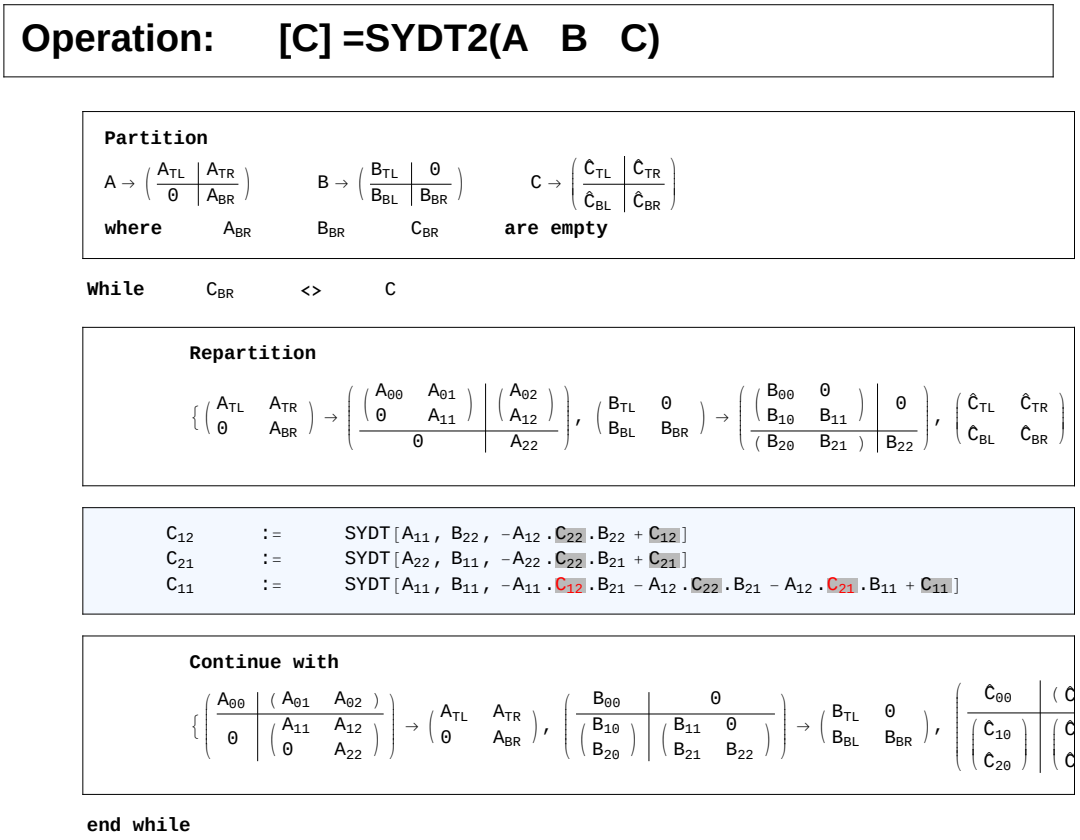

Figure 3.6: Mechanically derived blocked algorithm for computing the solution of the DTSY equation: loop-invariant (3.5).

Fig. 3.6 contains the algorithm, derived from this loop-invariant, as returned by our system. Predictably, the algorithms only differ in the updates, in Steps 6, 7 and 8 of the worksheet. Some variables in the updates are marked in gray or in red; the coloring refers to the availability of data at the moment in which the updates are executed. A gray quantity is presently available in the matrix. The boxes in first update

$$C_{12} := \mathrm{DTSY}(A_{11}, B_{22}, \boxed{C_{12}} - A_{12}\boxed{C_{22}}B_{22})$$

indicate that the two quantities needed in the assignment are available in submatrices C_{12} and C_{22} right after the repartitioning. Only submatrices of matrix C are highlighted, because C is the only matrix that is overwritten as the computation unfolds. Data from matrices A and B is available throughout the entire computation. Notice that the result of the first assignment is stored into submatrix C_{12}; this means that once that assignment is executed, the quantity C_{12} is not available anymore.

A red submatrix indicates that the quantity needed is not present in the submatrix after the repartitioning. It needs to be computed first to avoid redundant computations. The third statement

$$C_{11} := \text{DTSY}(A_{11}, B_{11}, C_{11} - A_{11} C_{12} B_{21} - A_{12} C_{22} B_{21} - A_{12} C_{21} B_{11})$$

refers to two red variables: C_{12} and C_{21}; these two submatrices need to be updated before their contents can be used as part of the third assignment. For this reason the updates are ordered so that C_{11} follows the assignments to C_{12} and C_{21}. Our mechanical system automatically identifies dependencies among the submatrices and orders the updates accordingly.

In Section 2 we completed the worksheet to derive one algorithm for computing the inverse of a triangular matrix (Fig. 3.7). We used the loop-invariant

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c} L_{TL}^{-1} & 0 \\ \hline -L_{BL}L_{TL}^{-1} & L_{BR} \end{array}\right). \tag{3.6}$$

In Fig. 3.7 we show the same worksheet, as generated by our system. The similarity is noticeable. Fig. 3.8 refers to the same derivation: it shows how the P_{after} is manipulated to find the updates (Step 8). The predicate in the figure corresponds to P_{after} after two passes of pattern matching: after the first pass P_{after} is expressed in terms of P_{before} (grey boxes), and in the second pass the system looks for quantities

Figure 3.7: Mechanically completed worksheet for the computation of the inverse of a triangular matrix: loop-invariant (3.6).

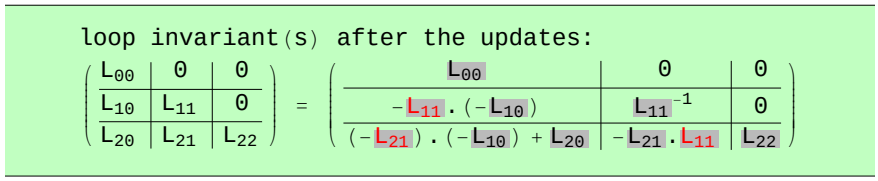

Figure 3.8: Inversion of a triangular matrix: The predicate P_{after} now contains the updates for loop-invariant (3.6).

common to different submatrices, so that computations can be saved (red boxes).

Finally, we consider a different loop-invariant for computing the inverse of a triangular matrix:

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c} L_{TL}^{-1} & 0 \\ \hline -L_{BR}^{-1} L_{BL} L_{TL}^{-1} & L_{BR} \end{array}\right). \tag{3.7}$$

The corresponding algorithm, mechanically generated, is displayed in Fig. 3.9. Again, the structure is the same as for the algorithm in Fig. 3.7, the difference being in the updates.

3.3 Scope and Limitations

Let $\mathcal{O}p$ be a target dense linear algebra operation and assume that the FLAME methodology for deriving algorithms applies to $\mathcal{O}p$. Evidence suggests that the family of algorithms resulting from the application of the derivation methodology to $\mathcal{O}p$ can be generated by a mechanical system. Our mechanical system, in particular, has been successfully employed in a number of different situations [7, 9, 6]:

- As part of the development of a full Level-3 BLAS library, a family of routines has been generated for each operation and for each parameter combination (transpose/no-transpose, upper/lower triangular, etc.). This project resulted in more than 300 mechanically derived algorithms. In addition, the system

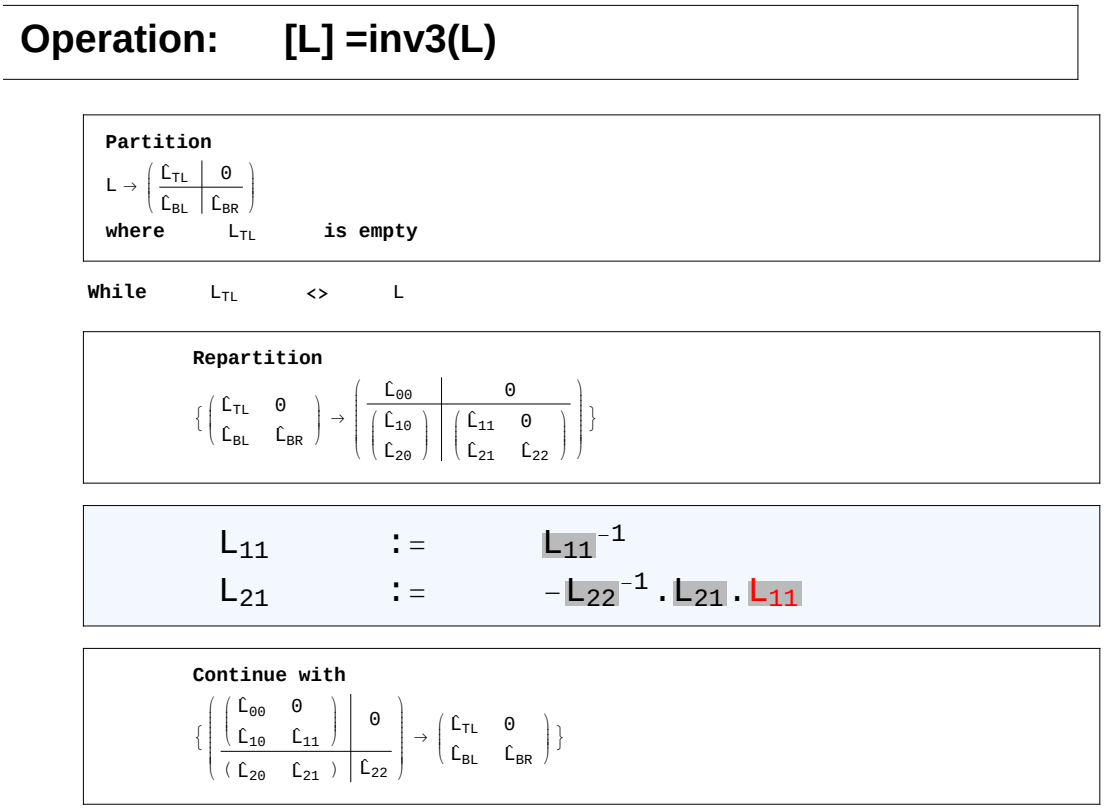

Figure 3.9: Mechanically derived blocked algorithm for inverting a triangular matrix: loop-invariant (3.7)

.

has been used to generate algorithms for the Cholesky and LU matrix factorizations. All the so-derived algorithms are available as part of a library.

- Computation of the covariance matrix. This operation has particular value to the Earth science and aerospace communities. In a collaborative effort, we derived families of algorithms for the operations required to compute the inverse of a symmetric positive definite matrix [6].

- Our prototype system has been extensively tested on control theory operations

from the RECSY library; we identified dozens of algorithms for each of these operations, including the particularly challenging triangular coupled Sylvester equation

$$\begin{cases} AX + YB = C, & A, B, D, E \text{ are triangular matrices,} \\ DX + YE = F, & X, Y \text{ are the unknowns,} \end{cases}$$

for which we found more than 50 algorithms.

At the same time we have encountered situations in which our system does not produce the desired results. In general, we observed the following two distinct scenarios.

- Pattern matching. The system correctly computes the predicates P_{before} and P_{after}, but it fails to find updates in terms of P_{before}. This is due to the fact that no canonical form has yet been identified for the symbolic expressions involved in the computation. The situation is best explained by an example. The next table shows possible expressions for the contents of P_{before} and P_{after} and the result of a pattern matching process (executed with Mathematica).

	P_{after}	P_{before}	Pattern Matching	Result
1)	(a c + b c)	a c	(a c + b c) /. {a c → **x**}	**x** + b c
2)	(a c + b c)	a + b	(a c + b c) /. {a + b → **x**}	a c + b c
3)	(a + b) c	a c	(a + b) c /. {a c → **x**}	(a + b) c
4)	(a + b) c	a + b	(a + b) c /. {a + b → **x**}	**x** c

  The "flat" representation (a c + b c) is suitable if we are looking for the expression (a c); by contrast, the factor a + b is found in (a c + b c) only if P_{after} is represented in "compact" form, as (a + b) c. This is a combinatorial problem. The QR factorization is one operation in which such a situation occurs.

- Operands with special properties. The matrices involved in a linear algebra computation might have a particular structure or property. When one such matrix is partitioned, some of these properties and structures are inherited by the submatrices. Example: If L is lower triangular and L is partitioned as $\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$ where L_{TL} is square, then L_{TL} and L_{BR} are also lower triangular. The current version of the prototype system does not have a mechanism to pass properties from a matrix to submatrices.

  Similarly, matrices with special structures may be traversed in a particular fashion. An example is given by the permutation matrix resulting from the LU factorization with pivoting; this matrix comes from the product of many permutation matrices with a distinctive structure. This structure needs to be encoded in order to mechanically derive algorithms for the LU factorization with pivoting.

3.4 Summary

We introduced a prototype mechanical system for deriving algorithms for dense linear algebra operations. The input to our prototype is a loop-invariant for the target operation together with the operands specifications. The output is either an algorithm description or an implementation. The system makes use of a database of PMEs for expressing the target operation in terms of simpler operations.

The following is a list of contributions made in this chapter.

- Necessary information. Each step of the derivation procedure is dictated by the input predicates P_{pre}, P_{post} and PME. We have shown that the information necessary for algorithms and routines to be generated mechanically is encoded in these three predicates.

- Mechanical derivation. Our mechanical systems requires little human inter-

vention. This attests to the systematic nature of our methodology for deriving formally correct algorithms, by demonstrating that it can be made mechanical.

- Linear algebra libraries. Our prototype system has been successfully applied to derive algorithms and routines for a large number of operations of different complexity. These range from BLAS3 operations and their variations, to complex control theory related operations, for which many previously unknown algorithms were discovered.

The ultimate goal is to let the users specify a target operation through an interface (or a website) and mechanically generate optimized dense linear algebra libraries at the push of the button. Our prototype system is an important advance towards this goal.

Chapter 4

Systematic Error Analysis

"One of the major difficulties in a practical analysis is that of description.
An ounce of analysis follows a pound of preparation."
B.N. Parlett [Matrix Eigenvalue Problems. Amer. Math. Monthly, 1965.]

Chapters 2 and 3 covered the problem of deriving families of algorithms for a target linear algebra operation. Our methodology guarantees that each algorithm in the family is formally correct, i.e., it computes the exact solution in a framework with exact arithmetic (a framework where no roundoff error occurs sometimes is also referred to as infinite precision arithmetic). The situation is different when computations are instead performed with finite precision arithmetic. Under these circumstances two formally correct algorithms may yield very different numerical results, possibly far from the exact solution to the problem. As a consequence, the numerical properties of every algorithm need to be investigated, in the attempt to set accuracy bounds that are always met by the computed solution. In Section 1.2.3 we have defined **stability** as a property that numerical algorithms need to possess in order to be considered useful.[1]

[1] Throughout the chapter we use the words "stability analysis" and "error analysis" interchange-

The study of stability properties of numerical algorithms is a challenging task. Quoting Higham [27]:

> "Two reasons why rounding error analysis can be hard to understand are that, first, there is no standard notation and, second, error analyses are often cluttered with re-derivations of standard results."

With our approach we also face another adversity: given a target operation, it is not uncommon for our methodology to produce families of a dozen or even more algorithms. While this abundance certainly represents one of the strengths of the FLAME methodology, at the same time it highlights the need for new tools, systematic if not mechanical, to approach the problem of identifying stable algorithms.

This chapter introduces extensions to the FLAME worksheet and to the procedure, presented respectively in Fig. 2.2 and Section 2.3, to perform stability analyses; the input for the procedure is now a target linear algebra operation together with a formula —to be proved— representing the desired stability result for the generated algorithm. The extended procedure consists of three stages: Stage 1 is the standard FLAME algorithm derivation; Stage 2 is a procedure similar to Stage 1 to prove a stability formula, and finally, in the third and last stage, error bounds are generated.

The primary two messages of this chapter follow.

1. The high level notation that abstracts from indices encourages and supports modular stability analysis. In the next sections we derive analyses incrementally, starting from basic operations which will constitute the building blocks for the analyses of subsequent operations.[2] We will provide multiple results for the same operation. This approach corresponds to building a library of known stability results, to be used as modules.

ably.

[2] In our treatment we only focus on the propagation of errors due to floating point arithmetic; we assume that situations of over-flow or under-flow do not occur.

2. The stability analysis of algorithms generated through our derivation methodology (or our mechanical system), can be made systematic; to this end we provide an extended derivation procedure to take into account the effects of inexact arithmetic.

The novelty of this chapter does not lie with the error analyses that we derive, but with the methodology for deriving the analyses. We present a framework and a procedure to be followed that demonstrate that a systematic approach to stability analysis is feasible. This is a first step towards an eventual mechanical system.

4.1 Extended Worksheet and Procedure

Starting from a description of a target linear algebra operation the goal is now not only to derive an algorithm for the operation, but also to prove a certain formula (given as input) related to the stability of the algorithm. Fig. 4.1 is an extended version of the FLAME worksheet (Fig. 2.2) which consists of two sides: the *Derivation* side on the left and the *Error* or *Stability* side on the right. The program statements (**Partition, While, Repartition, Continue**) are common to both sides. The field labeled "Error Analysis for the Updates" is the link between the derivation of an algorithm and its stability: it dictates how the analysis of the algorithm on the right side can be expressed in terms of the analysis of the updates on the left side.

4.1.1 The Derivation Side

The left side of the Fig. 4.1 is no different than the FLAME worksheet, with the exception that now we take roundoff error into account: the computation of a mathematical formula is not guaranteed to return the exact result. The immediate consequence is that the expressions that we used as loop-invariants are not mathematically correct anymore. We introduce notation to distinguish between exact and computed

Derivation side		Error side	
LA Operation		**Stability formula**	Step
Partition Operands **where**		Error Operands	3
{ Loop-Invariant }		{ Error Invariant }	2
while G **do**		{ }	4
Repartition Operands **where**		Error Operands	5a
{ Loop-Invariant }		{ Error Invariant }	6
Updates	Error Analysis for the Updates	Error Updates	8
{ Loop-Invariant }		{ Error Invariant }	7
Continue with Operands		Error Operands	5b
enddo			

Figure 4.1: Extended worksheet for proving stability analyses.

quantities; loop-invariants will then be re-expressed in terms of the computed results.

Notation:

- The function $fl\big(expression\big)$ returns the result of the evaluation of $expression$, where every operation is executed in floating point arithmetic.
  Note that $fl(x + y + z/w) = fl(fl(fl(x) + fl(y)) + fl(fl(z)/fl(w)))$.
- The notation $\big[expression\big]$ is shorthand for $fl\big(expression\big)$.
- $lhs = rhs$, denotes the equality relation between the quantities lhs and rhs.
- $lhs := rhs$, denotes the assignment $lhs \leftarrow rhs$. Formally, using a Hoare triple,

$\{True\}\ lhs := rhs\ \{lhs = [rhs]\}$.

- In the context of a program, the statements $lhs := rhs$ and $lhs := [rhs]$ are equivalent.

- Given an assignment $\kappa := exp$, we use the notation $\check{\kappa}$ to denote the quantity resulting from $fl(exp)$, which is actually stored in the variable κ.

- Given a vector $v \in \mathbb{R}^n$, $|v|$ corresponds to the vector $(|v_1|, \ldots, |v_n|)^T$. Similarly for matrices: $|A| = \begin{pmatrix} |\alpha_{11}| & \ldots & |\alpha_{1n}| \\ \vdots & \ddots & \\ |\alpha_{m1}| & \ldots & |\alpha_{mn}| \end{pmatrix}$, where $A \in \mathbb{R}^{m \times n}$.

- The notation $\delta\xi$, where the symbol δ is connected to a scalar variable ξ, indicates a perturbation associated to the variable ξ. Similarly, ΔX (Δ is connected with X) indicates a perturbation matrix associated to X. Vectors are special cases of a matrix.

In Chapter 2 we assumed that the computation was performed in exact arithmetic. In the setting of floating point arithmetic, the expressions that we used as loop-invariants are not appropriate anymore. Accurate formulae for the loop-invariants can be written by making use of the $\check{}$ and [ ] notations.

Example 3 ($\sum x$) *Let us consider the problem of adding all the entries of a vector x and accumulating the result in the variable κ; we denote this operation as $\kappa := \sum x$.*

Partitioning x as $\left(\frac{x_T}{x_B}\right)$, a possible loop-invariant for this operation is $\kappa = \sum x_T$, corresponding to the algorithm $\mathcal{A}_{Sum}$ that traverses the vector x from the top to the bottom, exposing at each iteration one or more entries from x_B and adding them up to the current contents of variable κ.

The loop-invariant expression $\kappa = \sum x_T$ is appropriate only if the entries of vector x are such that no error is introduced when the additions are performed

(assuming that no overflow occurs). The quantity stored in κ is then precisely $\sum x_T$. This is typically not true when x contains general floating point numbers.

Since floating point operations are inexact, the quantity stored in the variable κ is not $\sum x_T$, but $[\sum x_T]$; this is indicated by the expression $\check{\kappa} = [\sum x_T]$ which is read as "the variable κ contains the quantity $\check{\kappa}$ resulting from the execution of algorithm $\mathcal{A}_{Sum}$ to evaluate $\sum x_T$ using floating point arithmetic."

Example 4 (TRSV) *We examine the solution of a triangular system $Lx = b$. The operation is identified by the predicates*

$$\begin{aligned} P_{\text{pre}} : \quad \{ \quad & Size(L) = (m \times m) \ \wedge \ \text{LowerTriangular}(L) \ \wedge \\ & Size(b) = (m \times 1) \ \wedge \ Size(x) = (m \times 1) \ \wedge \ \text{Inv}(L) \ \wedge \\ & \text{Output}(L) \ \wedge \ x = \hat{x} \ \} \end{aligned}$$

and

$$P_{\text{post}} : \{ \ Lx = b \ \}, \quad \text{alternatively written as} : \{ \ x = \text{TRSV}(L, b) \ \}.$$

The PME for this operation is

$$\left(\frac{x_T = L_{TL}^{-1} b_T}{x_B = L_{BR}^{-1}(b_B - L_{BL} x_T)} \right),$$

or equivalently

$$\left(\frac{x_T = \text{TRSV}(L_{TL}, b_T)}{x_B = \text{TRSV}(L_{BR}, b_B - L_{BL} x_T)} \right).$$

A possible loop-invariant is

$$\left(\frac{x_T = L_{TL}^{-1} b_T}{x_B = \hat{x}_B} \right), \ \textit{which corresponds to} \ \left(\frac{x_T = \text{TRSV}(L_{TL}, b_T)}{x_B = \hat{x}_B} \right);$$

in the presence of roundoff, the accurate expressions for this loop-invariant are

$$\left(\frac{\check{x}_T = [L_{TL}^{-1} b_T]}{\check{x}_B = \hat{x}_B}\right), \text{ or } \left(\frac{\check{x}_T = [\text{TRSV}(L_{TL}, b_T)]}{\check{x}_B = \hat{x}_B}\right).$$

Formulating loop-invariants by means of ˇ-quantities and [ ] does not affect the derivation process discussed in Chapter 2 (a loop invariant remains a subset of the PME, in terms of executed operations). The modification merely regulates the relation between the loop-invariant and the variables used to store the computed result.

Assessing properties and bounds relative to ˇ-quantities ($\check{\kappa}$ and $\check{x}$ in the examples) is the goal of an error analysis. In the extended procedure we initiate such a study by performing the error analysis (central field in Step 8 of Fig. 4.1) for the updates from the Derivation side and propagating it to the Error side.

4.1.2 The Error Side

It is important to realize that the error analysis for a linear algebra operation algorithm is an operation itself, where the quantity to be computed is the matrix/operand expressing how error accumulates in the algorithm. We will call this matrix the *Error* matrix (or, more generally, operand). On the Error side, the FLAME methodology for deriving algorithms is applied to such an operation.

Example 5 (TRSV, continued) *An algorithm that computes the operation $Lx = b$ is said to be backward stable if the computed vector $\check{x}$ is the exact solution of a linear system "close" to the original one. Closeness here indicates small, possibly null, relative perturbations to the input operands L and b. It can be proved (and we provide a proof in this chapter) that the algorithms generated by the FLAME procedure for computing the solution of a triangular system are stable: the computed vector $\check{x}$ is the exact solution to a problem $(L+\Delta L)x = b$ where ΔL is a matrix whose*

entries represent small relative perturbations to L. The operation $(L + \Delta L)\check{x} = b$ represents the stability analysis for an algorithm computing the TRSV operation, and ΔL is the unknown error matrix.

The right side of the worksheet in Fig. 4.1, the Error side, has the same structure as the Derivation side, but deals with the operation that characterizes the error analysis for the algorithm appearing in the left side. We call Error Invariant a loop-invariant for this operation, referring to the fact that it is a loop-invariant for an operation expressing the result of an error analysis. The process of deriving the Error side can be described by the two alternatives:

1. finding an algorithm for computing the Error matrix, or
2. generating an inductive proof of the stability formula (input), Step 6 symbolizes the inductive hypothesis, Step 7 is the thesis and Step 8 contains the facts to proceed from the former to the latter.

Example 6 (TRSV, continued) *Let us assume the Derivation side of Fig. 4.1 had been completed already, generating algorithm $\mathcal{A}_{\text{TRSV}}$ for computing $Lx = b$. The formula we want to prove is $(L + \Delta L)\check{x} = b$, where ΔL is unknown. One algorithm for computing ΔL can be found by filling in the Error side: this same process can be also described as the construction of an inductive proof.*
The PME for the operation $(L + \Delta L)x = b$ is

$$\left(\frac{(L_{TL} + \Delta L_{TL})\check{x}_T = b_T}{(L_{BR} + \Delta L_{BR})\check{x}_B = b_B - L_{BL}\check{x}_T} \right),$$

and an error invariant we will use is

$$\left(\frac{(L_{TL} + \Delta L_{TL})\check{x}_T = b_T}{\check{x}_B = 0} \right).$$

We want the error invariant to be true *at the beginning and the end of each loop iteration. The core of the proof is in Steps 6, 7 and 8.*

{}		{ Error Invariant }	6R
Updates	Error Analysis for the Updates	Error Updates	8R
{}		{ Error Invariant }	7R

Step 6R *contains the error invariant before algorithm* $\mathcal{A}_{\text{TRSV}}$ *executes the updates: in this example, the expression*

$$\left(\begin{array}{c} (L_{00} + \Delta L_{00})\check{x}_0 = b_0 \\ \hline \check{\chi}_1 = 0 \\ \hline \check{x}_2 = 0 \end{array} \right) \tag{4.1}$$

encodes the assumption that the error invariant is currently satisfied, i.e., it is an inductive hypothesis. Step 7R *contains the expression for the error invariant that needs to be* true *after the execution of the computational updates:*

$$\left(\begin{array}{c} (L_{00} + \Delta L_{00})\check{x}_0 = b_0 \\ \hline (l_{10}^T \delta l_{10}^T)\check{x}_0 + (\lambda_{11}\delta\lambda_{11})\check{\chi}_1 = \beta_1 \\ \hline \check{x}_2 = 0 \end{array} \right); \tag{4.2}$$

in other words, it is the thesis to be proved.

We are seeking error updates (matrix ΔL *is the unknown) such that the error invariant is satisfied throughout the computation; this is possible if at each iteration, algorithm* $\mathcal{A}_{\text{TRSV}}$ *produces errors that can be accumulated into* ΔL*.*

Step 8 (left) contains the statements performed by the algorithm under consideration, in this case the assignment $\chi_1 := fl\left(\frac{\beta_1 - l_{10}^T x_0}{\lambda_{11}}\right)$*. A study of the errors*

introduced appears in the center field. The right field (Step 8R*) contains assignments necessary to collect the errors into the Error operand (ΔL). While here we are only outlining the procedure, Sec. 4.4 and Fig. 4.3 will provide the complete proof for the backward stability of the algorithm $\mathcal{A}_{\mathrm{TRSV}}$.*

4.1.3 Three Stages

Let us consider a target operation $\mathcal{OP}$ for which algorithms can be constructed by applying the methodology described in Chapter 2, and a stability formula $\mathcal{F}$.

Stage 1 of the extended procedure consists of generating one algorithm $\mathcal{A}$ computing $\mathcal{OP}$: this is achieved by filling in the Derivation side in Fig. 4.1. In Step 8 the updates for algorithm $\mathcal{A}$ are identified: these are the computational statements involving floating point operations; they produce inexact results.

In Stage 2 we try to accumulate the errors produced by the computational statements into the error matrix/operands. This is accomplished by

a) choosing an error analysis for each of the updates,

b) selecting an Error invariant for the stability formula, and

c) distributing the errors identified in a) into the error invariant to maintain its truth at the end of each iteration. If this is possible then the formula for the stability is proved, otherwise new choices for a) or b) can be attempted.

In Stage 3 numerical bounds for the error operands are established. Since Stage 2 provides a recursive definition for the error operands, the numerical bounds we are looking for can be expressed as the solution of a recurrence relation.

The main focus of this chapter is the propagation of errors, i.e., Stage 2. For the operations that we analyze we also provide a brief treatment of numerical bounds for the performed analyses (Stage 3).

4.2 Stability Analysis: Preliminaries

In the next sections we build up a small library of error results for linear algebra operations, focusing mainly on the second stage of the extended procedure. We start from the simplest operation, the inner product, to advance to more complex operations, making use of the analyses we have already proved. First we briefly introduce definitions and results regarding floating point arithmetic needed in this chapter.

1. Machine Precision: the quantity $\mathbf{u}$ is called *machine precision* or *unit roundoff* and normally is of the order of 10^{-8} and 10^{-16} for single and double precision arithmetic, respectively.[3] The unit roundoff is defined as the maximum positive floating point number which can be added to the number stored as 1 without changing the number stored as 1: $fl(1 + \mathbf{u}) = 1$.

2. Standard Computation Model (SCM): for any two floating point numbers x and y, we assume that the basic arithmetic operations satisfy the equality

$$\big[x \text{ op } y\big] = (x \text{ op } y)(1 + \epsilon), \qquad |\epsilon| \leq \mathbf{u}, \text{ and } \text{ op } \in \{+, -, *, /\}.$$

   The quantity ϵ is function of x, y and op. In the remainder of this chapter we will add a subscript $(\epsilon_+, \epsilon_*, \dots)$ to make clear what operation generated the $(1 + \epsilon)$ error factor. From now on we will assume that all the input variables are, or consist of, floating point numbers.

3. Alternative Computational Model (ACM) [27]: For certain problems it is convenient to also assume the following properties for the basic arithmetic oper-

[3] $\mathbf{u}$ is machine dependent; it is a function of the parameters characterizing the machine arithmetic: $\mathbf{u} = \frac{1}{2}\beta^{1-t}$, where β is the base and t is the precision of the floating point number system for the machine.

ations.

$$\big[x \text{ op } y\big] = \frac{x \text{ op } y}{1+\epsilon}, \qquad |\epsilon| \leq \mathbf{u}, \text{ and } \text{ op } \in \{+,-,*,/\}.$$

As for the standard computation model, the quantity ϵ is a function of x, y and op. Note that the ϵ's produced using the standard and alternative models are not necessarily equal.

4. When $n \in \mathbb{N}$ and $n\mathbf{u} < 1$, we define $\gamma_n := \dfrac{n\mathbf{u}}{1-n\mathbf{u}}$.
Although the factor γ_i appears in basically every error analysis, its subscript is of little use. Specifically, when working on error bounds, the exact value of the constant does not play a crucial role. Because of this observation the γ_i factors are often all rounded up to a common γ_n to facilitate grouping.

Lemma 1 *Let $\epsilon_i \in \mathbb{R}$, $0 \leq i \leq n-1$, and $|\epsilon_i| \leq \mathbf{u}$. Then $\exists\ \theta_n \in \mathbb{R}$ such that*

$$\prod_{i=0}^{n-1}(1+\epsilon_i)^{\pm 1} = 1+\theta_n, \quad \text{with} \quad |\theta_n| \leq \gamma_n.$$

Proof: See [27, Lemma 3.1]. $\diamond$

The quantity θ_n will be used throughout this chapter. For simplicity of expressions, it is not to be intended as a specific number, but as an order of magnitude identified by the subscript n. This subscript measures how many error factors of the form $(1+\epsilon_i)$ or $\frac{1}{(1+\epsilon_i)}$ are grouped together. **Two instances of the symbol θ_n, appearing even in the same expression, do not represent the same number!**

Lemma 2 (Distribution of Error) *Let α, β and λ be scalars and consider the assignment $\sigma := \dfrac{\alpha+\beta}{\lambda}$; the following three relations are satisfied:*

1. $\breve{\sigma} = \dfrac{(\alpha+\delta\alpha)+(\beta+\delta\beta)}{\lambda}$, *where* $|\delta\alpha| \leq \gamma_2|\alpha|$ *and* $|\delta\beta| \leq \gamma_2|\beta|$;
2. $\breve{\sigma} = \dfrac{\alpha+\beta}{(\lambda+\delta\lambda)}$, *where* $|\delta\lambda| \leq \gamma_2|\lambda|$;

3. $|\alpha + \beta - \check{\sigma}\lambda| = |\theta_2 \check{\sigma}\lambda| \leq \gamma_2 |\check{\sigma}||\lambda|$.

Hence $\check{\sigma}\lambda = \alpha + \beta + \mathcal{E}$, *with* $|\mathcal{E}| \leq \gamma_2 |\check{\sigma}||\lambda|$.

If in the above assignment $\lambda = 1$, *then the results hold with* γ_1 *in place of* γ_2.

This lemma states that the quantity $\check{\sigma}$ resulting from the evaluation of $\dfrac{\alpha+\beta}{\lambda}$ is the exact solution of close problems. It also indicates how the error generated in performing such an assignment can be accumulated in different ways.

Proof: Result 1. follows directly from the definition of STC (Preliminary #2), while 3. is an immediate consequence of 2. Here we provide a proof of 2.:

$$\begin{aligned}
\check{\sigma} &= \left[\frac{\alpha+\beta}{\lambda}\right] = \left[\frac{[\alpha+\beta]}{\lambda}\right] \\
&= \left[\frac{\alpha+\beta}{\lambda(1+\epsilon_+)}\right] && \text{ACM (Preliminary \#3)} \\
&= \frac{\alpha+\beta}{\lambda(1+\epsilon_+)(1+\epsilon_/)} && \text{ACM (Preliminary \#3)} \\
&= \frac{\alpha+\beta}{\lambda(1+\theta_2)} = \frac{\alpha+\beta}{\lambda+\delta\lambda} && \text{Lemma 1} \quad \diamond
\end{aligned}$$

4.3 Inner Product: $\kappa := x^T y$, $\kappa := \frac{\alpha + x^T y}{\lambda}$

We consider the operation $\kappa := x^T y$, where x and y are vectors in $\mathbb{R}^n$. Because of floating point arithmetic, any algorithm for computing κ returns a quantity $\check{\kappa}$ which is not necessarily equal to $x^T y$; the question is whether $\check{\kappa}$ is the exact solution for a nearby problem: $\check{\kappa} \stackrel{?}{=} x'^T y'$, where the vectors x' and y' are close to or equal to x and y.

Let $\mathcal{A}$ be the algorithm that computes the inner product $x^T y$ by sweeping x and y from the top down. The error analysis we will prove is expressed by the following theorem.

Theorem 1 *Let $x, y \in \mathbb{R}^n$, and let $\kappa := x^T y$ be computed by algorithm $\mathcal{A}$. Then $\check{\kappa} = \left[x^T y\right] = x^T \Delta y$, where*

$$\Delta = \Delta^{\{n\}} = I + \Theta^{\{n\}} = \begin{bmatrix} 1+\theta_n & & & & \\ & 1+\theta_n & & & \\ & & 1+\theta_{n-1} & & \\ & & & \ddots & \\ & & & & 1+\theta_2 \end{bmatrix}, \tag{4.3}$$

and $|\theta_i| \leq \gamma_i = \frac{i\mathbf{u}}{1-i\mathbf{u}}$ (Preliminary #4).

The superscript in $\Delta^{\{n\}}$ and $\Theta^{\{n\}}$ indicates the dimensions of the matrix and it is included only when the size is not obvious from the context. We remind that the θ_n quantities appearing in the matrix Δ do not represent the same number. The theorem says that the quantity $\check{\kappa}$ is the exact solution for two problems: $\kappa := x'^T y$, where $x'^T = x^T \Delta$ and $\kappa := x^T y'$, where $y' = \Delta y$. In both cases, the input vectors (x' and y') are close to the original inputs (x and y).

We present two different proofs for this theorem. The first is a standard proof by induction, while the second is obtained by filling in the FLAME extended worksheet (Fig. 4.1).

Proof: Standard Approach (induction).

Base case. $n = 1$: the vectors x and y are scalars; from the definition of the standard computation model (Preliminary # 2) and using Lemma 1:

$$[x^T y] = xy(1+\epsilon_*) = x(1+\theta_1)y.$$

Inductive step. Inductive hypothesis ($n = k$): assume $x, y \in \mathbb{R}^k$, then $[x^T y] = x^T \Delta^{\{k\}} y$. The result for $n = k+1$ must be proved: if two vectors $x, y \in \mathbb{R}^{k+1}$,

then $\left[x^T y\right] = x^T \Delta^{\{k+1\}} y$. Partition vectors x and y as

$$x \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right),\ y \rightarrow \left(\begin{array}{c} y_0 \\ \hline \mu_1 \end{array}\right)$$

where $x_0, y_0 \in \mathbb{R}^k$ and χ_1 and μ_1 are scalars. The thesis can be rewritten as $[x^T y] = \left[[x_0^T y_0] + [\chi_1^T \mu_1]\right]$. Using the inductive hypothesis first and then Preliminary # 2 (twice)

$$\begin{array}{rcl} [x^T y] & = & \left[x_0^T \Delta^{\{k\}} y_0 + [\chi_1^T \mu_1]\right] \\ & = & \left[x_0^T \Delta^{\{k\}} y_0 + \chi_1^T (1+\epsilon_*)\mu_1\right] \\ & = & x_0^T \Delta^{\{k\}} (1+\epsilon_+) y_0 + \chi_1^T (1+\epsilon_*)(1+\epsilon_+)\mu_1 \end{array}$$

which can be written as

$$[x^T y] = \left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right)^T \left(\begin{array}{c|c} \Delta^{\{k\}}(1+\epsilon_+) & \\ \hline & (1+\epsilon_*)(1+\epsilon_+) \end{array}\right) \left(\begin{array}{c} y_0 \\ \hline \mu_1 \end{array}\right).$$

The equality (from Lemma 1)

$$\Delta^{\{k+1\}} = \left(\begin{array}{c|c} \Delta^{\{k\}}(1+\epsilon_+) & \\ \hline & (1+\epsilon_*)(1+\epsilon_+) \end{array}\right)$$

leads to the formula $[x^T y] = x^T \Delta^{\{k+1\}} y$, which proves the theorem. $\diamond$

Proof: New Approach (FLAME extended worksheet).

In the standard approach we only gave a sketch of the algorithm $\mathcal{A}$ computing $\kappa := x^T y$; in Stage 1 of the extended procedure the algorithm is formally framed within the FLAME worksheet. This is done in the left side of Fig. 4.2. The loop guard and the partitioned (and repartitioned) variables apply to the Error side too.

The theorem can be posed as the solution of the equation $\check{\kappa} = x^T \Delta y$, where

the diagonal matrix Δ is the unknown. Such an equation corresponds to a linear algebra operation for which algorithms can be derived using the FLAME methodology: a PME and a valid loop-invariant for this operation can be identified. The PME is

$$\check{\kappa} = \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)^T \left(\begin{array}{c|c} \Delta_T & \\ \hline & \Delta_B \end{array}\right) \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = x_T \Delta_T \, y_T + x_B \Delta_B \, y_B$$

and $\check{\kappa} = x_T^T \Delta_T \, y_T$ is a viable error invariant. Such an error invariant can now be entered on the Error side of the worksheet to derive one algorithm (Fig. 4.2).

We already know from the Derivation side that $\mathcal{A}$ proceeds by traversing the vectors x and y top down. The partitioning and repartitioning of matrix Δ follows conformally. The important steps of the derivation are Steps 6R through 8R. Step 6R is the Error invariant at the top of the iteration, expressed in terms of repartitioned operands; it represents the theorem's inductive hypothesis: at the top of the loop $\check{\kappa}$ can be written as $x_0^T \Delta_0 y_0$, in other words, we are assuming that the computation performed so far produced an error that can be accumulated into a diagonal matrix Δ_0. The question is whether the same structure can be maintained after the execution of the statement in Step 8.

Step 7 contains the expression for the loop-invariant that needs to be true at the bottom of the iteration; it characterizes the thesis to be proved:
$\check{\kappa} = x_0^T \Delta_0^{\{k\}} y_0 + \chi_1 \delta_1 \psi_1$, or in other words, after the execution of the statement in Step 8, the error generated in the computation can still be accumulated into a diagonal matrix. Here the matrix $\Delta_0^{\{k\}}$ and the scalar δ_1 are the unknowns.

It remains to 1) analyze the error produced by $\kappa := \kappa + \chi_1 \psi_1$, and 2) find a way for assigning the error to the variables $\Delta_0^{\{k\}}$ and δ_1. The error generated executing $\kappa := \kappa + \chi_1 \psi_1$ is easily determined by applying the formula from the

Derivation side		Error side	
$\kappa := x^T y$		$\check{\kappa} = x^T \Delta y$	Step
Partition $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right), y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$ **where** x_T is empty		$\Delta \rightarrow \left(\begin{array}{c\|c} \Delta_T & 0 \\ \hline 0 & \Delta_B \end{array}\right)$	3
$\left\{\check{\kappa} = \left[x_T^T y_T\right]\right\}$		$\left\{\check{\kappa} = x_T^T \Delta_T y_T\right\}$	2
while $m(x_B) > 0$ **do**		$\{k = m(x_T)\}$	4
Repartition $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right), \left(\begin{array}{c\|c} \Delta_T & 0 \\ \hline 0 & \Delta_B \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} \Delta_0 & 0 & 0 \\ \hline 0 & \delta_1 & 0 \\ \hline 0 & 0 & \Delta_2 \end{array}\right)$ **where**			5a
$\left\{\check{\kappa} = \left[x_0^T y_0\right]\right\}$		$\left\{\check{\kappa} = x_0^T \Delta_0 y_0 = x_0^T \Delta_0^{\{k\}} y_0\right\}$	6
$\kappa := \kappa + \chi_1 \psi_1$	$\check{\kappa} = \big(\check{\kappa} + \chi_1\psi_1(1+\epsilon_*)\big)(1+\epsilon_+)$ $= x_0^T \Delta_0^{\{k\}}(1+\epsilon_+) y_0 +$ $\chi_1(1+\epsilon_*)(1+\epsilon_+)\psi_1$	$\Delta_0 := \Delta_0(1+\epsilon_+)$ $\delta_1 := (1+\epsilon_+)(1+\epsilon_*)$	8
$\left\{\check{\kappa} = \left[x_0^T y_0 + \chi_1\psi_1\right]\right\}$		$\left\{\begin{array}{l} \check{\kappa} = \left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right)^T \left(\begin{array}{c\|c} \Delta_0^{\{k\}} & \\ \hline & \delta_1 \end{array}\right) \left(\begin{array}{c} y_0 \\ \hline \psi_1 \end{array}\right) = \\ \qquad = x_0^T \Delta_0^{\{k\}} y_0 + \chi_1 \delta_1 \psi_1 \end{array}\right\}$	7
Continue with $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right), \left(\begin{array}{c\|c} \Delta_T & 0 \\ \hline 0 & \Delta_B \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} \Delta_0 & 0 & 0 \\ \hline 0 & \delta_1 & 0 \\ \hline 0 & 0 & \Delta_2 \end{array}\right)$			5b
enddo			

Figure 4.2: Extended worksheet completed to prove the backward stability of the Inner Product.

standard computation model (Preliminary # 2), twice:

$$\begin{aligned} \check{\kappa} &= [\check{\kappa} + \chi_1\psi_1] = \big[\check{\kappa} + [\chi_1\psi_1]\big] \\ &= \big[\check{\kappa} + \chi_1\psi_1(1+\epsilon_*)\big] \\ &= \big(\check{\kappa} + \chi_1\psi_1(1+\epsilon_*)\big)(1+\epsilon_+). \end{aligned}$$

It is now time to use the inductive hypothesis from Step 6 $\check{\kappa} = x_0^T \Delta_0 y_0 = x_0^T \Delta_0^{\{k\}} y_0$; replacing $\check{\kappa}$, we get

$$\check{\kappa} = x_0^T \Delta_0^{\{k\}} (1+\epsilon_+) y_0 + \chi_1 (1+\epsilon_*)(1+\epsilon_+)\psi_1$$

from which it is straightforward to identify the assignments

$$\Delta_0 := \Delta_0 (1+\epsilon_+)$$

$$\delta_1 := (1+\epsilon_+)(1+\epsilon_*)$$

that maintain the Error invariant *true* at the bottom of the iteration. This concludes the proof that $[x^T y]$, computed by algorithm $\mathcal{A}$, can be written as $x^T \Delta y$ where Δ is a suitable diagonal matrix. The complete extended worksheet for algorithm $\mathcal{A}$ together with the proof of its analysis $\check{\kappa} = [x^T y]$ is shown in Fig. 4.2.

The third stage of the procedure concerns finding quantitative bounds for the error matrix. In this case, the magnitude of the entries of matrix Δ is easily determined thanks to the following inductive relation; we omit the formal steps.

$$\Delta^{\{k+1\}} = \left(\begin{array}{c|c} \Delta^{\{k\}}(1+\epsilon_+) & \\ \hline & (1+\epsilon_*)(1+\epsilon_+) \end{array} \right) \diamond$$

Since $\check{\kappa} = [x^T y] = x^T \Delta y = x^T y + x^T \Theta y$, it follows that $\check{\kappa} = (x^T + \delta x^T) y = x^T (y + \delta y)$, where the vectors δx and δy contain small relative perturbations (entry-wise) of vectors x and y respectively.

Theorem 1 allows us to state results concerning the backward and forward stability of the inner product, independently of the order of evaluation.

Corollary 1 (Backward Analysis).

a) $fl(x^T y) = x'^T y$, *with* $x' = (x + \delta x)^T$ *and* $\delta x^T = x^T \Theta$. *Independently of the*

order of evaluation, $|\delta x| \leq \gamma_n |x|$ *holds.*

b) $fl(x^T y) = x^T y'$*, with* $y' = (y + \delta y)$ *and* $\delta y = \Theta y$*. Independently of the order of evaluation,* $|\delta y| \leq \gamma_n |y|$ *holds.*

Corollary 2 (Forward Analysis). *For any order of evaluation:*

$$|\kappa - \check{\kappa}| = |x^T y - fl(x^T y)| \leq \gamma_n |x|^T |y| \text{ or}$$
$$\check{\kappa} = \kappa + \mathcal{E}, \text{ where } |\mathcal{E}| \leq \gamma_n |x|^T |y|.$$

The last two corollaries can be grouped together to state how the error produced in the computation of $\kappa := x^T y$ can be accumulated into the quantities $\check{k}, x$ and y.

Lemma 3 (Distribution of Error). *Let* $x, y \in \mathbb{R}^n$*, and* $\kappa := x^T y$*. The following equalities are satisfied:*

1. $\check{\kappa} = (x + \delta x)^T y$*, with* $|\delta x| \leq \gamma_n |x|$*;*
2. $\check{\kappa} = x^T (y + \delta y)$*, with* $|\delta y| \leq \gamma_n |y|$*;*
3. $\check{\kappa} + \mathcal{E}_{\check{\kappa}} = x^T y$*, with* $|\mathcal{E}_{\check{\kappa}}| \leq \gamma_n |x|^T |y|$*;*

The following theorem represents the analysis for the assignment $\sigma := \dfrac{\alpha + \beta}{\lambda}$ (already considered in Lemma 2), in the special case when β corresponds to an inner product $-x^T y$. Notice that upon completion, in exact arithmetic, the quantity α would equal $v^T z$ where $v = \begin{pmatrix} x \\ \check{\lambda} \end{pmatrix}$ and $z = \begin{pmatrix} y \\ \kappa \end{pmatrix}$. Because of roundoff error, the equality does not hold; the theorem also gives a measure of the distance between α and the computed $v^T z$.

Theorem 2 *Let* α *and* λ *be scalars,* x *and* y *be vectors in* $\mathbb{R}^{k-1}$*; consider the assignment* $\kappa := \dfrac{\alpha - x^T y}{\lambda}$ *and the quantity* $\check{\kappa}$ *computed as* $\frac{(\alpha - (x^T y))}{\lambda}$*. Then:*

1. $\check{\kappa} = \dfrac{\alpha - (x + \delta x)^T y}{(\lambda + \delta\lambda)}$*, with* $\delta x = \Theta^{\{k-1\}} x$ *(matrix* Θ *is defined in Eqn. (4.3)), and* $\delta\lambda = \theta_2 \lambda$*.*

In general, independently of the order in which the factor $(\alpha - x^T y)$ *is computed:*

2. $\check{\kappa} = \dfrac{\alpha - (x + \delta x)^T y}{(\lambda + \delta\lambda)}$, *with* $|\delta x| \leq \gamma_k |x|$, *and* $|\delta\lambda| = \gamma_k |\lambda|$;

3. $|\alpha - x^T y - \check{\kappa}\lambda| \leq \gamma_k (|x|^T |y| + |\check{\kappa}||\lambda|) = \gamma_k |v|^T |z|$, *where* $v = \begin{pmatrix} x \\ \check{\lambda} \end{pmatrix}$ *and* $z = \begin{pmatrix} y \\ \kappa \end{pmatrix}$.

Moreover:

4. $\check{\kappa} = \dfrac{(\alpha + \delta\alpha) - (x + \delta x)^T y}{\lambda}$, *where* $|\delta\alpha| \leq \gamma_k |\alpha|$ *and* $|\delta x| \leq \gamma_{k+1} |x|$, *and* $\check{\kappa} = \dfrac{(\alpha + \delta\alpha) - (x + \delta x)^T y}{(\lambda + \delta\lambda)}$, *where* $|\delta\alpha| \leq \gamma_{k-1} |\alpha|$, $|\delta x| \leq \gamma_k |x|$ *and* $|\delta\lambda| \leq \gamma_1 |\lambda|$.

The theorem holds true *(with minor modifications to the* γ *factors) also when there is no division in the assignment (*$\lambda = 1$*).*

Proof:

1.
$$\begin{aligned} \check{\kappa}_1 &= \left[\frac{\alpha - x^T y}{\lambda_{11}} \right] \\ &= \left[\frac{[\alpha - [x^T y]]}{\lambda_{11}} \right] \\ &= \left[\frac{[\alpha - x^T \Delta^{\{k-1\}} y]}{\lambda_{11}} \right] && \text{Theorem 1} \\ &= \frac{\alpha - x^T \Delta y}{\lambda_{11}(1 + \theta_2)} && \text{Lemma 2 (Part 2)} \end{aligned}$$

   The thesis is obtained using Eqn. (4.3) and expanding.

2. For the ordering considered in Part 1, the proof is straightforward. The opposite ordering, corresponding to the inner product of vectors $v = \begin{pmatrix} \alpha \\ -x \end{pmatrix}$ and $z = \begin{pmatrix} 1 \\ y \end{pmatrix}$, is more tedious. A sketch of the proof follows:

$$\check{\kappa} = \left[\frac{[v^T z]}{\lambda} \right] = \left[\frac{v^T \Delta^{\{k\}} z}{\lambda} \right],$$

where the top-left entry of Δ is actually $1 + \theta_{k-1}$; v, z and Δ are partitioned to expose the first entry. The proof is completed by dividing by the first entry of Δ and using Lemma 1.

3. Using Part 2 of this theorem,

$$\check{\kappa}\lambda + \check{\kappa}\delta\lambda = \alpha - x^T y - \delta x^T y,$$

from which

$$|\alpha - \check{\kappa}\lambda - x^T y| = |\check{\kappa}\delta\lambda + \delta x^T y| \leq \gamma_k(|\lambda||\kappa| + |x^T||y|).$$

4. A particular ordering in the computation of $(\alpha - x^T y)$ corresponds to the product of the vectors

$$\left(\begin{array}{c} -x_T \\ \hline \alpha \\ \hline -x_B \end{array}\right)^T \left(\begin{array}{c} y_T \\ \hline 1 \\ \hline y_B \end{array}\right),$$

where $x = \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ and $y = \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$. A direct application of Theorem 1 to these vectors proves the thesis.

$\diamond$

4.4 TRSV: $Lx = b$

Next, we analyze the stability of the algorithms for computing the solution of the triangular system $Lx = b$. A formal specification for the operation is provided in Examples 4 through 6 in this chapter. The goal is to prove that the algorithms for computing the vector x are backward stable, i.e., the solution $\check{x}$ satisfies the equality $(L + \Delta L)\check{x} = (b + \delta b)$, where ΔL and δb contain small relative perturbations of the

entries of L and b respectively. Thus, $(L + \Delta L)\check{x} = (b + \delta b)$ is the stability formula that we want to prove. In fact, we show that the vector or perturbation δb is zero, meaning that the roundoff error can be entirely accumulated into the matrix ΔL.

The algorithm corresponding to loop-invariant

$$\left(\begin{array}{c} x_T = L_{TL}^{-1} b_T \\ \hline x_B = \hat{x}_B \end{array} \right)$$

is derived in the left side of Fig. 4.3. Such an algorithm computes the vector x from the top to the bottom, keeping the top part of x fully computed and the bottom part untouched. At each iteration it performs the numerical update (Step 8), $\chi_1 := \dfrac{\beta_1 - l_{10}^T x_0}{\lambda_{11}}$, which is the statement that introduces errors due to floating point arithmetic.

On the Error side we consider the error invariant

$$\left(\begin{array}{c} (L_{TL} + \Delta L_{TL})\check{x}_T = (b_T + \delta b_T) \\ \hline \check{x}_B = 0 \end{array} \right)$$

which corresponds to the fact that the top portion of x that has been computed so far is the solution of a nearby problem. In Steps 6 and 7 the invariant is expressed in terms of the repartitioned operands:

$$\left(\begin{array}{c} (L_{00} + \Delta L_{00})\check{x}_0 = (b_0 + \delta b_0) \\ \hline \check{\chi}_1 = 0 \\ \hline \check{x}_2 = 0 \end{array} \right), \quad \left(\begin{array}{c} (L_{00} + \Delta L_{00})\check{x}_0 = (b_T + \delta b_0) \\ \hline (l_{10}^T \delta l_{10}^T)\check{x}_0 + (\lambda_{11} \delta\lambda_{11})\check{\chi}_1 = (\beta_T + \delta\beta_1) \\ \hline \check{x}_2 = 0 \end{array} \right).$$

We seek error updates (Step 8R) to transition from the former to the latter, thus proving that the error invariant is *true* at the beginning and at the end of each iteration. The two expressions for the error invariant differ only in the row relative to $\check{\chi}$: this is a natural consequence of the fact that the update in Step 8 (left) assigns

Derivation side	Error side	Step
$Lx = b$	$(L + \Delta L)\check{x} = b$	
Partition $L \rightarrow \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right), b \rightarrow \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right)$, $\Delta L \rightarrow \left(\begin{array}{c\|c} \Delta L_{TL} & 0 \\ \hline \Delta L_{BL} & \Delta L_{BR} \end{array}\right)$ **where** L_{TL} is $0 \times 0, \ldots$		3
$\left\{ \left(\begin{array}{c} \check{x}_T = [L_{TL}^{-1} b_T] \\ \hline \check{x}_B = 0 \end{array} \right) \right\}$	$\left\{ \left(\begin{array}{c} (L_{TL} + \Delta L_{TL})\check{x}_T = b_T \\ \hline \check{x}_B = 0 \end{array} \right) \right\}$	2
while $m(x_B) > 0$ **do**	$\{k = m(x_T)\}$	4
Repartition $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right)$, $\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right)$, $\left(\begin{array}{c\|c} \Delta L_{TL} & 0 \\ \hline \Delta L_{BL} & \Delta L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} \Delta L_{00} & 0 & 0 \\ \hline \delta l_{10}^T & \delta\lambda_{11} & 0 \\ \hline \Delta L_{20} & \delta l_{21} & \Delta L_{22} \end{array}\right)$ **where**		5a
$\left\{ \left(\begin{array}{c} \check{x}_0 = [L_{00}^{-1} b_0] \\ \hline \check{\chi}_1 = 0 \\ \hline \check{x}_2 = 0 \end{array} \right) \right\}$	$\left\{ \left(\begin{array}{c} (L_{00} + \Delta L_{00})\check{x}_0 = b_0 \\ \hline \check{\chi}_1 = 0 \\ \hline \check{x}_2 = 0 \end{array} \right) \right\}$	6
$\chi_1 := \dfrac{\beta_1 - l_{10}^T x_0}{\lambda_{11}}$	$\check{\chi}_1 \lambda_{11}(1 + \theta_2) = \beta_1 - l_{10}^T (I + \Theta)\check{x}_0$	$\delta l_{10}^T := l_{10}^T \Theta$ $\delta\lambda_{11} := \theta_2 \lambda_{11}$	8
$\left\{ \left(\begin{array}{c} \check{x}_0 = [L_{00}^{-1} b_0] \\ \hline \check{\chi}_1 = [\lambda_{11}^{-1}(\beta_1 - l_{10}^T \check{x}_0)] \\ \hline \check{x}_2 = 0 \end{array} \right) \right\}$	$\left\{ \left(\begin{array}{c} (L_{00} + \Delta L_{00})\check{x}_0 = b_0 \\ \hline (l_{10}^T + \delta l_{10}^T)\check{x}_0 + (\lambda_{11} + \delta\lambda_{11})\check{\chi}_1 = \beta_1 \\ \hline \check{x}_2 = 0 \end{array} \right) \right\}$	7
Continue with $\ldots$	$\left(\begin{array}{c\|c} \Delta L_{TL} & 0 \\ \hline \Delta L_{BL} & \Delta L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} \Delta L_{00} & 0 & 0 \\ \hline \delta l_{10}^T & \delta\lambda_{11} & 0 \\ \hline \Delta L_{20} & \delta l_{21} & \Delta L_{22} \end{array}\right)$	5b
enddo		

Figure 4.3: $Lx = b$: FLAME extended worksheet for proving the backward stability of variant 1.

a new value to χ_1 only. Before the execution of the update we know (inductive hypothesis) that $\check{\chi}_1 = 0$; we are left to prove that after the execution the relation (expanded)

$$\lambda_{11}\check{\chi}_1 + \delta\lambda_{11}\check{\chi}_1 + l_{10}^T\check{x}_0 + \delta l_{10}^T\check{x}_0 = \beta_1 + \delta\beta_1. \tag{4.4}$$

is also *true*. Remember that the unknown is the matrix ΔL_{TL} and specifically the variables δl_{10}^T and $\delta\lambda_{11}$ here.

The analysis for the update in Step 8, $\chi_1 := \dfrac{\beta_1 - l_{10}^T x_0}{\lambda_{11}}$, is given in Part 1 of Theorem 2:

$$\check{\chi} = \frac{\beta_1 - l_{10}^T\Delta x_0}{\lambda_{11}(1+\theta_2)},$$

and since $\Delta = I + \Theta$, expanding,

$$\lambda_{11}\check{\chi}_1 + \lambda_{11}\theta_2\check{\chi}_1 + l_{10}^T\check{x}_0 + l_{10}^T\Theta\check{x}_0 = \beta_1, \tag{4.5}$$

which appears in the central field of Step 8. From a comparison between equations 4.5 and 4.4 it is recognized that the assignments $\delta l_{10}^T := l_{10}^T\Theta$, $\delta\lambda_{11} := \theta_2\lambda_{11}$ make the error invariant in Step 7R *true*. This proves that the error invariant is maintained *true* at the beginning and the end each iteration of the algorithm, thus after completion, too, implying the stability formula for this operation. Moreover, since neither δb_0 nor $\delta\beta_1$ are ever assigned, we conclude that the error vector δb is identically zero, resulting in the formula $(L + \Delta L)\check{x} = b$. To avoid clutter, in Fig. 4.3 we do not display the vector δb.

A few words on the magnitude of the entries of matrix ΔL. The diagonal entries are set by the assignment $\delta\lambda_{11} := \theta_2\lambda_{11}$, while the sub-diagonal entries are set one row at a time by means of the assignment $\delta l_{10}^T := l_{10}^T\Theta$. Therefore matrix

$\Delta L =$ has the following structure

$$\Delta L = \begin{bmatrix} \theta_2\lambda_{11} & & & & & & \\ \theta_1\lambda_{21} & \theta_2\lambda_{22} & & & & & \\ \theta_2\lambda_{31} & \theta_2\lambda_{32} & \theta_2\lambda_{33} & & & & \\ \theta_3\lambda_{41} & \theta_3\lambda_{42} & \theta_2\lambda_{43} & \theta_2\lambda_{44} & & & \\ \vdots & \vdots & \vdots & & \ddots & & \\ \theta_{i-1}\lambda_{i1} & \theta_{i-1}\lambda_{i2} & \theta_{i-2}\lambda_{i3} & & & \theta_2\lambda_{ii} & \\ \vdots & \vdots & \vdots & & & & \ddots \\ \theta_{n-1}\lambda_{n1} & \theta_{n-1}\lambda_{n2} & \theta_{n-2}\lambda_{n3} & \cdots & & \theta_2\lambda_{n(n-1)} & \theta_2\lambda_{nn} \end{bmatrix} \tag{4.6}$$

representing small relative entry-wise perturbations of the of matrix L (λ_{ij} is the (ij) entry of matrix L). A convenient entry-wise bound for matrix $\Delta L =$ is $|\Delta L| \leq \theta_n |L|$.

The next theorems establish the backward stability of all the algorithms derived by the FLAME methodology.

Theorem 3 *Let $\mathcal{A}_{\mathrm{TRSV1}}$ be the algorithm for solving $Lx = b$ identified by the loop-invariant* $\left(\dfrac{\check{x}_T = [L_{TL}^{-1} b_T]}{\check{x}_B = \hat{x}_B} \right)$. *Then the vector $\check{x}$ computed by algorithm $\mathcal{A}_{\mathrm{TRSV1}}$ satisfies the equality*

$$(L + \Delta L)\check{x} = b$$

where matrix ΔL is as in 4.6; therefore $|\Delta L| \leq \theta_n |L|$.

Proof: See Fig. 4.3. $\diamond$

Theorem 4 *Let $\mathcal{A}_{\mathrm{TRSV2}}$ be the algorithm for solving $Lx = b$ identified by the loop-invariant* $\left(\dfrac{\check{x}_T = [L_{TL}^{-1} b_T]}{\check{x}_B = [-L_{BL}\check{x}_T]} \right)$. *Then the vector $\check{x}$ computed by algorithm $\mathcal{A}_{\mathrm{TRSV2}}$ satisfies the equality $(L + \Delta L)\check{x} = b$, where $|\Delta L| \leq \theta_n |L|$.*

Theorem 5 *Let $\mathcal{A}_{\mathrm{TRSV3}}$ be the algorithm for solving $Lx = b$ identified by the*

loop-invariant $\left(\dfrac{\check{x}_T = [L_{TL}^{-1} b_T]}{\check{x}_B = [b_B - L_{BL}\check{x}_T]}\right)$. *Then the vector* $\check{x}$ *computed by algorithm* $\mathcal{A}_{\mathrm{TRSV3}}$ *satisfies the equality* $(L + \Delta L)\check{x} = b$, *where* $|\Delta L| \leq \theta_n |L|$.

These last two theorems are proved with analogous techniques. We omit the worksheets corresponding to these proofs.

Finally we state a corollary that will be used in the analysis of the LU factorization.

Corollary 3 *The vector* $\check{x}$ *computed by any of the algorithmic variants mentioned in the three theorems above satisfies* $|L\check{x} - b| = |\Delta L\check{x}| \leq \gamma_n |L||\check{x}|$.

4.5 LU Factorization: $LU = A$

We are now ready to tackle the analysis of the LU factorization. The operation is defined as

$$\begin{aligned} P_{\mathrm{pre}} : \quad \{ \quad & Size(A) = (m \times m) \ \wedge \ \mathrm{Output}(L, U) \ \wedge \\ & Size(L) = (m \times m) \ \wedge \ \mathrm{UnitLowerTriangular}(L) \ \wedge \ L = 0 \ \wedge \\ & Size(U) = (m \times m) \ \wedge \ \mathrm{UpperTriangular}(U) \ \wedge \ U = 0 \ \wedge \ \ldots \ \} \end{aligned}$$

and

$$P_{\mathrm{post}} : \{ \ LU = A \ \}, \quad \text{alternatively written as} : \big\{ \{L \backslash U\} = \mathrm{LU}(A) \ \big\},$$

where the predicate UnitLowerTriangular(L) is *true* when the matrix L is lower triangular with ones on the diagonal. The dots indicate that the conditions on

matrix A to ensure the existence of the LU factorization are missing. The PME is

$$\left(\begin{array}{c|c} \{L\backslash U\}_{TL} = \mathrm{LU}(A_{TL}) & U_{TR} = L_{TL}^{-1}A_{TR} \\ \hline L_{BL} = A_{BL}U_{TL}^{-1} & \{L\backslash U\}_{BR} = \mathrm{LU}(A_{BR} - L_{BL}U_{TR}) \end{array}\right),$$

where the notation $\{L\backslash U\}_{xy} = \mathrm{LU}(M)$ signifies that matrices L_{xy} and U_{xy} are the result of the LU factorization of matrix M (here $xy = TL$ or BR). In this section we focus our attention on the loop-invariant

$$\left(\begin{array}{c|c} \{\check{L}\backslash\check{U}\}_{TL} = [\mathrm{LU}(A_{TL})] & \check{U}_{TR} = [\check{L}_{TL}^{-1}A_{TR}] \\ \hline \check{L}_{BL} = [A_{BL}\check{U}_{TL}^{-1}] & \{\check{L}\backslash\check{U}\}_{BR} = 0 \end{array}\right)$$

corresponding to the Crout variant $\mathcal{A}_{\mathrm{LU1}}$ for computing the LU factorization of a matrix A. The derivation of the algorithm is in Fig. 4.4. The information that we need for the error analysis is given by the three updates

$$\mu_{11} := \alpha_{11} - l_{10}^T u_{01}, \qquad u_{12}^T := a_{12}^T - l_{10}^T U_{02}, \qquad l_{21} := \frac{a_{21} - L_{20}u_{01}}{\mu_{11}}.$$

The goal is to show that algorithm $\mathcal{A}_{\mathrm{LU1}}$ is backward stable, in the sense that the computed factors $\check{L}$ and $\check{U}$ satisfy the equality $\check{L}\check{U} = A + \Delta A$: they are the exact L and U factors of a matrix close to A.

In Fig. 4.5, we show only the error side of the extended worksheet (to save space) with the error invariant

$$\left(\begin{array}{c|c} \{\check{L}\check{U}\}_{TL} = (A_{TL} + \Delta A_{TL}) & \check{U}_{TR} = \check{L}_{TL}^{-1}(A_{TR} + \Delta A_{TR}) \\ \hline \check{L}_{BL} = (A_{BL} + \Delta A_{BL})\check{U}_{TL}^{-1} & \{\check{L}\check{U}\}_{BR} = 0 \end{array}\right),$$

showing what the inductive hypothesis and thesis are (Step 6 and Step 7). Matrix ΔA is the error operand: it represents the unknown where we want to accumulate the errors produced by the updates. In order to complete the error analysis we need to find error updates for this matrix.

We start by comparing the expressions for the error invariant at Step 6 and

Step	$LU = A$
3	**Partition** $L \rightarrow \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), U \rightarrow \left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right), A \rightarrow \left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ **where** $L_{TL}, U_{TL}, A_{TL},$ are 0×0
4	**While** $m(A_{BR}) > 0$ **do**
5a	**Repartition** $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right), \left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \mu_{11} & u_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right)$ **where**
6	$\left\{\left(\begin{array}{c\|c\|c} \{\check{L}\backslash\check{U}\}_{00} = [\mathrm{LU}(A_{00})] & \check{u}_{01} = [\check{L}_{00}^{-1} a_{01}] & \check{U}_{02} = [\check{L}_{00}^{-1} A_{02}] \\ \hline \check{l}_{10}^T = [a_{10}^T \check{U}_{00}^{-1}] & \check{\mu}_{11} = 0 & \check{u}_{12}^T = 0 \\ \hline \check{L}_{20} = [A_{20} \check{U}_{00}^{-1}] & \check{l}_{21} = 0 & \{\check{L}\backslash\check{U}\}_{22} = 0 \end{array}\right)\right\}$
8	$\mu_{11} := \alpha_{11} - l_{10}^T u_{01}$ $u_{12}^T := a_{12}^T - l_{10}^T U_{02}$ $l_{21} := \dfrac{a_{21} - L_{20} u_{01}}{\mu_{11}}$
7	$\left\{\left(\begin{array}{c\|c\|c} \{\check{L}\backslash\check{U}\}_{00} = [\mathrm{LU}(A_{00})] & \check{u}_{01} = [\check{L}_{00}^{-1} a_{01}] & \check{U}_{02} = [\check{L}_{00}^{-1} A_{02}] \\ \hline \check{l}_{10}^T = [a_{10}^T \check{U}_{00}^{-1}] & \check{\mu}_{11} = [\alpha_{11} - \check{l}_{10}^T \check{u}_{01}] & \check{u}_{12}^T = [a_{12}^T - \check{l}_{10}^T \check{U}_{02}] \\ \hline \check{L}_{20} = [A_{20} \check{U}_{00}^{-1}] & \check{l}_{21} = [(a_{21} - \check{L}_{20} \check{u}_{01}) / \check{\mu}_{11}] & \{\check{L}\backslash\check{U}\}_{22} = 0 \end{array}\right)\right\}$
5b	**Continue with** $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right), \left(\begin{array}{c\|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \mu_{11} & u_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right)$
	endwhile

Figure 4.4: $LU = A$. Left side of the FLAME extended worksheet: Crout variant.

Step 7: they differ only in submatrices $(1,1)$ μ_{11}, $(1,2)$ u_{12}^T and $(2,1)$ l_{21}. It is not by accident: the computational statements update the same three quadrants.

Position $(1,1)$**:** The element μ_{11} is only affected by the statement $\mu_{11} := \alpha_{11} - l_{10}^T u_{01}$. The appropriate form for the analysis of this statement is evident when looking at the difference between the inductive thesis and the inductive hypothesis: the hypothesis tells us that μ_{11} has no error associated with it (in fact

Derivation side	Error side
$LU = A$	$\check{L}\check{U} = (A + \Delta A)$
Partition $\Delta A \rightarrow \left(\begin{array}{c\|c} \Delta A_{TL} & \Delta A_{TR} \\ \hline \Delta A_{BL} & \Delta A_{BR} \end{array}\right)$ **where**	
$\left\{\left(\begin{array}{c\|c} \{\check{L}\check{U}\}_{TL} = (A_{TL} + \Delta A_{TL}) & \check{U}_{TR} = \check{L}_{TL}^{-1}(A_{TR} + \Delta A_{TR}) \\ \hline \check{L}_{BL} = (A_{BL} + \Delta A_{BL})\check{U}_{TL}^{-1} & \{\check{L}\check{U}\}_{BR} = 0 \end{array}\right)\right\}$	
while $m(A_{BR}) > 0$ **do**	$\{k = m(A_{TL})\}$
Repartition $\left(\begin{array}{c\|c} \Delta A_{TL} & \Delta A_{TR} \\ \hline \Delta A_{BL} & \Delta A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} \Delta A_{00} & \delta a_{01} & \Delta A_{02} \\ \hline \delta a_{10}^T & \delta\alpha_{11} & \delta a_{12}^T \\ \hline \Delta A_{20} & \delta a_{21} & \Delta A_{22} \end{array}\right)$ **where**	
$\left\{\left(\begin{array}{c\|c\|c} \{\check{L}\backslash\check{U}\}_{00} = \mathrm{LU}(A_{00}+\Delta A_{00}) & \check{u}_{01} = \check{L}_{00}^{-1}(a_{01}+\delta a_{01}) & \check{U}_{02} = \check{L}_{00}^{-1}(A_{02}+\Delta A_{02}) \\ \hline \check{l}_{10}^T = (a_{10}^T+\delta a_{10}^T)\check{U}_{00}^{-1} & \check{\mu}_{11} = 0 & \check{u}_{12}^T = 0 \\ \hline \check{L}_{20} = (A_{20}+\Delta A_{02})\check{U}_{00}^{-1} & \check{l}_{21} = 0 & \{\check{L}\backslash\check{U}\}_{22} = 0 \end{array}\right)\right\}$		
$\mu_{11} := \alpha_{11} - l_{10}^T u_{01}$ $u_{12}^T := a_{12}^T - l_{10}^T U_{02}$ $l_{21} := \dfrac{a_{21} - L_{20}u_{01}}{\mu_{11}}$	$\check{\mu}_{11} = \alpha_{11} - \check{l}_{10}^T\check{u}_{01} + \mathcal{E}_{11}$ $\check{u}_{12}^T = a_{12}^T - \check{l}_{10}^T\check{U}_{02} + \mathcal{E}_{12}^T$ $\check{l}_{21}\check{\mu}_{11} = a_{21} - \check{L}_{20}\check{u}_{01} + \mathcal{E}_{21}$	$\delta\alpha_{11} = \mathcal{E}_{11}$ $\delta a_{12}^T = \mathcal{E}_{12}^T$ $\delta a_{21} = \mathcal{E}_{21}$
$\left\{\left(\begin{array}{c\|c\|c} \{\check{L}\backslash\check{U}\}_{00} = \mathrm{LU}(A_{00}+\Delta A_{00}) & \check{u}_{01} = \check{L}_{00}^{-1}(a_{01}+\delta a_{01}) & \check{U}_{02} = \check{L}_{00}^{-1}(A_{02}+\Delta A_{02}) \\ \hline \check{l}_{10}^T = (a_{10}^T+\delta a_{10}^T)\check{U}_{00}^{-1} & \check{\mu}_{11} = (\alpha_{11}+\delta\alpha_{11}) - \check{l}_{10}^T\check{u}_{01} & \check{u}_{12}^T = (a_{12}^T+\delta a_{12}^T) - \check{l}_{10}^T\check{U}_{02} \\ \hline \check{L}_{20} = (A_{20}+\Delta A_{20})\check{U}_{00}^{-1} & \check{l}_{21} = (a_{21}+\delta a_{21} - \check{L}_{20}\check{u}_{01})/\check{\mu}_{11} & \{\check{L}\backslash\check{U}\}_{22} = 0 \end{array}\right)\right\}$		
Continue with $\left(\begin{array}{c\|c} \Delta A_{TL} & \Delta A_{TR} \\ \hline \Delta A_{BL} & \Delta A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} \Delta A_{00} & \delta a_{01} & \Delta A_{02} \\ \hline \delta a_{10}^T & \delta\alpha_{11} & \delta a_{12}^T \\ \hline \Delta A_{20} & \delta a_{21} & \Delta A_{22} \end{array}\right)$		
enddo		

Figure 4.5: $LU = A$. Right side of the extended worksheet for proving the backward stability of the LU factorization computed via the Crout variant. Step 6 contains the inductive hypothesis and Step 7 the thesis. The center field of Step 8 contains the error analysis for the computational updates and the right field shows the assignments that maintain the error invariant *true*.

its value is still 0), while the thesis

$$\check{\mu}_{11} = \alpha_{11} - \check{l}_{10}^T \check{u}_{01} + \delta\alpha_{11}.$$

indicates that once the update is performed, $\check{\mu}_{11}$ contains the quantity $\alpha_{11} - \check{l}_{10}^T \check{u}_{01}$ plus the error term $\delta\alpha_{11}$. Thus, $\delta\alpha_{11}$ is the difference between $\check{\mu}_{11}$ and $\alpha_{11} - \check{l}_{10}^T \check{u}_{01}$. A bound for such difference is given in Theorem 2 (Part 3, with $\lambda = 1$): $\mathcal{E}_{11} = \check{\mu}_{11} - \alpha_{11} + \check{l}_{10}^T \check{u}_{01}$, and $|\mathcal{E}_{11}| \leq \gamma_k \left| \begin{pmatrix} \check{l}_{10} \\ 1 \end{pmatrix} \right|^T \left| \begin{pmatrix} \check{u}_{01} \\ \check{\mu}_{11} \end{pmatrix} \right|$, where $k = m(A_{TL})$.

Position $(1, 2)$**:** Submatrix u_{12}^T is updated by the assignment $u_{12}^T := a_{12}^T - l_{10}^T U_{02}$. The inductive thesis requires the error to be accumulated into vector δa_{12}^T. The analysis is the same as the one for position $(1, 1)$, given that every element of u_{12}^T is updated the same way that μ_{11} is. The vector δa_{12}^T is the difference $\mathcal{E}_{12}^T$ between $\check{u}_{12}^T$ and $a_{12}^T - \check{l}_{10}^T \check{U}_{02}$. Applying Part 3 of Theorem 2, we conclude that an entry-wise bound for $|\mathcal{E}_{12}^T|$ is $\gamma_k \begin{pmatrix} |\check{l}_{10}| \\ 1 \end{pmatrix}^T \begin{pmatrix} |\check{U}_{02}| \\ |\check{u}_{12}^T| \end{pmatrix}$.

Position $(2, 1)$**:** The error invariant for quadrant l_{21} dictates that

$$\check{l}_{21} = \frac{a_{21} + \delta a_{21} - \check{L}_{20} \check{u}_{01}}{\check{\mu}_{11}},$$

which means that δa_{21} is the difference $\mathcal{E}_{21}$ between the vectors $\check{l}_{21}\check{\mu}_{11}$ and $\check{L}_{20}\check{u}_{01} - a_{21}$. Part 3 of Theorem 2 gives us an entry-wise bound for vector $\mathcal{E}_{21}$:

$$|\mathcal{E}_{21}| \leq \gamma_k \left(\begin{array}{c|c} |\check{L}_{20}| & |\check{l}_{21}| \end{array} \right) \begin{pmatrix} |\check{u}_{01}| \\ |\check{\mu}_{11}| \end{pmatrix}.$$

A comment on the size of the entries of matrix $\Delta\mathcal{A}$ follows. At each iterate, the computational updates on Step 8 guarantee that the following relations are

satisfied

$$\begin{aligned} \alpha_{11} &= l_{10}^T u_{01} + \mu_{11}, \\ a_{12}^T &= l_{10}^T U_{02} + u_{12}^T, \\ a_{21} &= L_{20} u_{01} + l_{21} \mu_{11}, \end{aligned} \tag{4.7}$$

so that upon termination, the loop computes matrices L and U such that $LU = A$. Conversely, at each iteration the Error updates compute the matrix ΔA that satisfies

$$\begin{aligned} \delta\alpha_{11} = \mathcal{E}_{11} \;\wedge\; |\mathcal{E}_{11}| &\leq \gamma_k \left(|\check{l}_{10}^T||\check{u}_{01}| + |\check{\mu}_{11}| \right), \\ \delta a_{12}^T = \mathcal{E}_{12}^T \;\wedge\; |\mathcal{E}_{12}^T| &\leq \gamma_k \left(|\check{l}_{10}^T||\check{U}_{02}| + |\check{u}_{12}^T| \right), \\ \delta a_{21} = \mathcal{E}_{21} \;\wedge\; |\mathcal{E}_{12}| &\leq \gamma_k \left(|\check{L}_{20}||\check{u}_{01}| + |\check{l}_{21}||\check{\mu}_{11}| \right), \end{aligned} \tag{4.8}$$

where k ranges from 1 to $n = n(A)$ and the $\leq$ relation holds entry-wise. From a comparison of Eqns. (4.7) and (4.8) it is clear that if we ignored the γ_k factors, matrix $|\Delta A|$ would be entry-wise bounded by $|\check{L}||\check{U}|$. In general, taking the γ_k's into account, $|\Delta A| \leq \gamma_n |\check{L}||\check{U}|$.

Theorem 6 *Let $\mathcal{A}_{\mathrm{LU1}}$ be the algorithm for computing the LU factorization identified via the loop-invariant*

$$\left(\begin{array}{c|c} \{\check{L}\backslash\check{U}\}_{TL} = [\mathrm{LU}(A_{TL})] & \check{U}_{TR} = [\check{L}_{TL}^{-1} A_{TR}] \\ \hline \check{L}_{BL} = [A_{BL} \check{U}_{TL}^{-1}] & \{\check{L}\backslash\check{U}\}_{BR} = 0 \end{array} \right),$$

which is also known as the Crout variant. Then the computed factors $\check{L}$ and $\check{U}$ are such that:

$$\check{L}\check{U} = A + \delta A$$

where $|\delta A| \leq \gamma_n |\check{L}||\check{U}|$.

Proof: From Figs. 4.4, 4.5. $\diamond$

Note: the backward stability result we just proved, $\check{L}\check{U} = A + \delta A$ with $|\delta A| \leq \gamma_n |\check{L}||\check{U}|$, agrees with the one that Nick Higham proves in [27]. In the

analysis of the updates (central field of Step 8 in Fig. 4.5) we could have used Part 4 of Theorem 2 (the error is accumulated in $\delta\alpha$ too) instead of Part 3. In that case we would have obtained the slightly weaker bound

$$|\delta A| \leq \gamma_n(|A| + |\check{L}||\check{U}|),$$

which is the result that Golub and Van Loan present in [20].

It is well known that there exist five algorithmic variants to compute the LU factorization; our methodology captures them all: the PME for this operation admits five feasible loop-invariants. Every treatment of the backward stability for the LU factorization is specific to one variant only, although similar bounds result.

We conclude the chapter by proving that the "bordered" variant for computing the LU factorization is also backward stable, and satisfies the same bounds as the Crout Variant. The computational updates for this variant include the solution of a triangular system. As a consequence we will make use of the stability results that we have proved in the former section (in particular Corollary 3). The stability of the other three variants can be proved by applying similar techniques.

Theorem 7 *Let $\mathcal{A}_{\mathrm{LU2}}$ be the algorithm for computing the LU factorization identified by loop-invariant*

$$\left(\begin{array}{c|c} \{\check{L}\backslash\check{U}\}_{TL} = [\mathrm{LU}(A_{TL})] & \check{U}_{TR} = 0 \\ \hline \check{L}_{BL} = 0 & \{\check{L}\backslash\check{U}\}_{BR} = 0 \end{array}\right),$$

which is also known as the "bordered" or "lazy" variant. Then the computed factors $\check{L}$ and $\check{U}$ satisfy

$$\check{L}\check{U} = A + \delta A,$$

where $|\delta A| \leq \gamma_n|\check{L}||\check{U}|$.

Proof: The core of the proof is given in Fig. 4.7; the error invariant is

$$\left(\begin{array}{c|c} \{\check{L}\check{U}\}_{TL} = (A_{TL} + \Delta A_{TL}) & \check{U}_{TR} = 0 \\ \hline \check{L}_{BL} = 0 & \{\check{L}\backslash\check{U}\}_{BR} = 0 \end{array}\right).$$

Fig. 4.6 shows the derivation side of the extended worksheet. $\diamond$

4.6 Summary

We presented extensions to the worksheet introduced in Chapter 2 that support error analyses of algorithms derived with our methodology. This chapter makes the following contributions to the field of numerical analysis.

- An alternative notation that hides explicit loop indexing.
- Raising the level of abstraction at which to reason about dense linear algebra algorithms leads to modularity. We showed that the stability analyses of simple operations constitute a library of building blocks for the analyses of more complex operations.
- Systematic error analysis. The investigation of numerical properties can be made systematic by employing the same classical computer science concepts that lead to the systematic derivation of algorithms.

These advances provide evidence that, for a class of dense linear algebra operations, we can hope to make stability analysis mechanical.

Derivation side | Error side

$LU = A$

Partition

$$L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right), A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad \ldots$$

where $L_{TL}, U_{TL}, A_{TL},$ are 0×0

$$\left\{\left(\begin{array}{c|c} \{\check{L}\backslash\check{U}\}_{TL} = [LU(A_{TL})] & \check{U}_{TR} = 0 \\ \hline \check{L}_{BL} = 0 & \{\check{L}\backslash\check{U}\}_{BR} = 0 \end{array}\right)\right\} \quad \{\}$$

while $m(A_{BR}) > 0$ **do** {}

Repartition

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \mu_{11} & u_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right), \quad \ldots$$

where

$$\left\{\left(\begin{array}{c|c|c} \{\check{L}\backslash\check{U}\}_{00} = [LU(A_{00})] & \check{u}_{01} = 0 & \check{U}_{02} \\ \hline \check{l}_{10}^T = 0 & \check{\mu}_{11} = 0 & \check{u}_{12}^T \\ \hline \check{L}_{20} = 0 & \check{l}_{21} = 0 & \{\check{L}\backslash\check{U}\}_{22} = 0 \end{array}\right)\right\} \quad \{\}$$

$u_{01} := L_{00}^{-1} a_{01}$

$l_{10}^T := a_{10}^T U_{00}^{-1}$

$\mu_{11} := \alpha_{11} - l_{10}^T u_{01}$

$$\left\{\left(\begin{array}{c|c|c} \{\check{L}\backslash\check{U}\}_{00} = [LU(A_{00})] & \check{u}_{01} = [\check{L}_{00}^{-1} a_{01}] & \check{U}_{02} \\ \hline \check{l}_{10}^T = [a_{10}^T \check{U}_{00}^{-1}] & \check{\mu}_{11} = [\alpha_{11} - \check{l}_{10}^T \check{u}_{01}] & \check{u}_{12}^T \\ \hline \check{L}_{20} = 0 & \check{l}_{21} = 0 & \{\check{L}\backslash\check{U}\}_{22} = 0 \end{array}\right)\right\} \quad \{\}$$

Continue with

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \mu_{11} & u_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right)$$

enddo

Figure 4.6: $LU = A$: FLAME extended worksheet for proving the backward stability of the LU factorization (bordered variant). Fig. 4.7 contains Steps 6, 7 and 8.

		$\left\{\left(\begin{array}{c\|c\|c} \{\check{L}\backslash\check{U}\}_{00} = \mathrm{LU}(A_{00}+\Delta A_{00}) & \check{u}_{01} = 0 & \check{U}_{02} = 0 \\ \hline \check{l}_{10}^T = 0 & \check{\mu}_{11} = 0 & \check{u}_{12}^T = 0 \\ \hline \check{L}_{20} = 0 & \check{l}_{21} = 0 & \{\check{L}\backslash\check{U}\}_{22} = 0 \end{array}\right)\right\}$	6R
$u_{01} := L_{00}^{-1} a_{01}$ TRSM $l_{10}^T := a_{10}^T U_{00}^{-1}$ TRSM $\mu_{11} := \alpha_{11} - l_{10}^T u_{01}$	$\check{L}_{00}\check{u}_{01} = a_{01} - \Delta\check{L}_{00}\check{u}_{01}$ (Corollary 3) $\implies \mathcal{E}_{01} = -\Delta\check{L}_{00}\check{u}_{01} \ \wedge\ \lvert\mathcal{E}_{01}\rvert \leq \gamma_k \lvert\check{L}_{00}\rvert\lvert\check{u}_{01}\rvert$ $\check{l}_{10}^T\check{U}_{00} = a_{10}^T - \check{l}_{10}^T\Delta\check{U}_{00}$ (Corollary 3) $\implies \mathcal{E}_{10}^T = -\check{l}_{10}^T\Delta\check{U}_{00} \ \wedge\ \lvert\mathcal{E}_{10}^T\rvert \leq \gamma_k \lvert\check{l}_{10}^T\rvert\lvert\check{U}_{00}\rvert$ $\check{\mu}_{11} = \alpha_{11} - \check{l}_{10}^T\check{u}_{01} + \mathcal{E}_{11} \wedge$ (Theorem 2) $\lvert\mathcal{E}_{11}\rvert \leq \gamma_k \left(\frac{\lvert\check{l}_{10}\rvert}{1}\right)^T \left(\frac{\lvert\check{u}_{01}\rvert}{\lvert\check{\mu}_{11}\rvert}\right)$	$\delta a_{01} = \mathcal{E}_{01}$ $\delta a_{10}^T = \mathcal{E}_{10}^T$ $\delta\alpha_{11} = \mathcal{E}_{11}$	8R
	$\left\{\left(\begin{array}{c\|c\|c} \{\check{L}\backslash\check{U}\}_{00} = \mathrm{LU}(A_{00}+\Delta A_{00}) & \check{u}_{01} = \check{L}_{00}^{-1}(a_{01}+\delta a_{01}) & \check{U}_{02} = 0 \\ \hline \check{l}_{10}^T = (a_{10}^T+\delta a_{10}^T)\check{U}_{00}^{-1} & \check{\mu}_{11} = (\alpha_{11}+\delta\alpha_{11}) - \check{l}_{10}^T\check{u}_{01} & \check{u}_{12}^T = 0 \\ \hline \check{L}_{20} = 0 & \check{l}_{21} = 0 & \{\check{L}\backslash\check{U}\}_{22} = 0 \end{array}\right)\right\}$		7R

Figure 4.7: $LU = A$: Zoom on Steps 6, 7 and 8 of the extended worksheet filled in to prove the backward stability of the bordered variant for computing the LU factorization. The derivation side of the worksheet is shown in Fig. 4.6. In the center field of Step 8 we also included bounds for the quantities assigns to the error matrix.

Chapter 5

Conclusions

In this thesis we focused on the derivation and the analysis of formally correct algorithms for dense linear algebra operations. Our goal was to achieve systematic derivation and analysis of algorithms. Succeeding in this goal represents a significant step forward towards a more ambitious project: the development of a mechanical system for the generation of linear algebra libraries. The inputs for such a mechanical system would be the mathematical specifications of a dense linear algebra operation, a programming language, and (a model of) a target architecture. At the press of a button, the system would return:

- A family of formally correct algorithms that compute the operation;
- A corresponding family of routines, implemented in the language of choice and optimized for the target architecture;
- Documentation for each algorithm in the family, consisting of stability and performance analyses.

The primary contribution of this thesis is to provide evidence that such a mechanical system can, to a large degree, be achieved. The results of our research and references

to our research publications are presented in the next section. We conclude with a discussion on potential future research directions.

5.1 Results

Here we summarize the main contributions of this thesis within the field of computer sciences.

- **Systematic derivation of formally correct algorithms** [4, 5]. Proving the formal correctness of loop-based algorithms is a process that involves the determination of a loop-invariant. No general procedure is known to determine loop-invariants. We showed that for a class of linear algebra operations, given the mathematical specification of the operation, it is possible to identify —*a priori*— a family of loop-invariants for algorithms that compute the operation. This result was facilitated by the choice of an appropriate level of abstraction, and a corresponding notation, for dealing with dense linear algebra algorithms [10]. This enabled us to develop a systematic procedure for building one algorithm corresponding to a given loop-invariant by applying formal derivation techniques and by exploiting the structure of dense linear algebra algorithms. The formal correctness of the resulting algorithm is guaranteed by construction [9]. Since there is a one-to-one correspondence between loop-invariant and algorithm, our methodology yields a family of algorithms for a given operation. In brief, we introduced a systematic procedure for deriving families of formally correct algorithms for dense linear algebra operations.

- **Mechanical generation of formally correct algorithms** [7]. A procedure is truly systematic if it can be executed by a mechanical system. Moreover, the execution of the derivation procedure by hand leads to complex matrix operations; such a process is error prone. We presented a prototype of a mechanical

system that implements the aforementioned derivation procedure. The system operates with limited human intervention: it takes a loop-invariant as input and returns a formally correct algorithm. Alternatively, it outputs a routine that, by means of high-level APIs [8], mirrors the algorithm description. This prototype system further demonstrates that our derivation procedure is systematic. Since we also reasoned that loop-invariants can be mechanically derived from the operation specifications, we conclude that for a class of dense linear algebra operations, formally correct algorithms can be generated mechanically.

- **Systematic stability analysis of algorithms** [9]. We have established that formally correct algorithms can be systematically and even mechanically generated. Unfortunately, since we are dealing with floating point computations, formal correctness does not translate into accuracy. A stability analysis of every generated algorithm is needed. To this end we extended the derivation procedure to investigate numerical properties. The procedure is systematic and modular: the error analysis for one algorithm builds upon the analyses for the linear algebra operations appearing in the loop-body of the algorithm.

This thesis also contributes to the field of computational science, through the algorithms and libraries it enables.

- **Families of algorithms.** A library should provide the users with many algorithmic variants for computing the same operation. In the introduction we illustrated that top performance is attained by different variants in different scenarios. Our methodology returns a family of algorithms (in some cases a family may include dozens of variants). As an example of successful collaboration, our prototype mechanical system was used to derive families of algorithms for all the operations necessary for computing the covariance matrix [6]; this project was motivated by research in Earth science and aerospace

engineering. The system has also been applied to several operations included in LAPACK.

- **Libraries.** Ultimately, computational scientists need a library. If a library included a family of algorithmic variants for every operation, as we advocate, its development would represent a daunting task for a human. Mechanical generation of routines is necessary. As part of the development of a complete set of BLAS3 programs, our prototype system was employed to generate more than 300 routines. The system has also been shown to apply to all the operations included in RECSY, a library targeting control theory applications [29, 30]; for these operations, our methodology typically yields dozens of variants.[1]

Thus, the generation —systematic and mechanical— of formally correct algorithms is the contribution of this dissertation to computer science, while the algorithms and libraries generated by our methodology and tools represent the contribution to the computational sciences and engineering.

5.2 Future Work

Here we propose possible directions to expand and strengthen the results achieved in this dissertation.

- **Mechanical derivation.** Our prototype system does not fully implement the procedure for deriving algorithms. It leaves the task of selecting loop-invariants to the user. The limitations of our prototype system have been discussed in Chapter 3. In order to expand the scope of the system, matrices with other structures (in addition to triangular and symmetric) should be incorporated. We believe that our system could be improved, addressing

[1] RECSY normally includes only three variants per operation.

both these issues (derivation of loop-invariants and structured matrices), by explicitly keeping track of the partitioning sizes.

- **Performance analysis.** Having a family of algorithms at hand is disorienting for the user who simply wants to attain the maximum performance on a target architecture.

  In Chapter 4 we have shown how the derivation procedure can be extended to study stability properties. We believe that the procedure can be extended in a similar fashion to study performance. The same way a loop-invariant describes the content of the variables during the computation, there could be a cost-invariant which holds at the same points where the loop-invariant does. The cost for the algorithm would then be expressed as a recurrence relation for which Mathematica can, in principle, find a closed form solution.

  A different approach for determining the best performing algorithmic variant in a specific situation would be by developing an intelligent system. Such a system would be responsible to run and time the different variants for the target operation and for the operations appearing in the loop-bodies. This would result in combinatorial tree to be traversed and pruned by using artificial intelligence techniques.

- **SCOPE.** We have established that the PME contains the necessary information for a mechanical system in order to generate algorithms. We would like to investigate the class of operations that admit a PME. The ideas employed in the derivation procedure are not restricted to the field of linear algebra only. We have mentioned already that formally correct algorithms for sorting can be derived with a similar methodology. The same is true for problems in computational geometry. A precise characterization of the scope of the methodology is missing.

- **Mechanical stability analysis.** The most ambitious part in the mechanical development of linear algebra libraries lies with the mechanical derivation of stability analysis. This dissertation provides evidence that for a class of operations, a user-suggested result can be proved systematically. Whether a stability formula can be assessed mechanically is an open question. One of the limitations of the methodology is that the execution of the extended procedure leads to complicated matrix expressions. This opens up the possibility of mechanical error analysis.

Appendix A

Cholesky Factorization

A.1 Derivation and FLAME Notation

Here we illustrate the FLAME notation [10] by deriving and representing algorithms to compute the Cholesky factorization of a matrix. Given a symmetric positive definite matrix A, the operation is defined as the computation of a lower triangular matrix L such that $LL^T = A$. We denote this operation by $A := \Gamma(A)$, which should be read as "the matrix A is overwritten by its Cholesky factor L." Since A is symmetric, typically only the lower or upper triangular part is stored, and it is that part that is then overwritten with the result. In this discussion, we assume that the lower triangular part of A is stored and overwritten.

One algorithm for computing $A := \Gamma(A)$ can be derived as follows. Consider the equation $A = LL^T$ and partition the matrices A and L as

$$A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right). \tag{A.1}$$

The $\star$ signifies that the symmetric part of A is neither stored nor updated.

We adopt the commonly used convention that Greek lower case letters refer to scalars, lower case letters refer to vectors, and upper case letters refer to matrices.

Substituting the partitioned matrices (A.1) into the equation $A = LL^T$, we find

$$\left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right)^T = \left(\begin{array}{c|c} \lambda_{11}^2 & \star \\ \hline \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{array}\right),$$

from which we conclude

$$\left(\begin{array}{c|c} \lambda_{11} = \sqrt{\alpha_{11}} & \star \\ \hline l_{21} = a_{21}/\lambda_{11} & L_{22} = \Gamma(A_{22} - l_{21} l_{21}^T) \end{array}\right).$$

These equalities motivate the following algorithm.

Algorithm 1 (Cholesky_unblocked).

1. Partition $A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right)$.
2. Overwrite $\alpha_{11} := \lambda_{11} = \sqrt{\alpha_{11}}$.
3. Overwrite $a_{21} := l_{21} = a_{21}/\lambda_{11}$.
4. Overwrite $A_{22} := A_{22} - l_{21} l_{21}^T$ (updating only the lower triangular part of A_{22}).
5. Continue with $A = A_{22}$. (Back to Step 1).

Pictorially the algorithm can be described by a sequence of pictures as shown in Fig. A.1.

a) The matrix is seen as composed by four quadrants: $A = \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$, where T, B, L, R signify *T*op, *B*ottom, *L*eft, *R*ight, respectively. Quadrant A_{TR} is not labelled because it is not stored and it is never referenced or

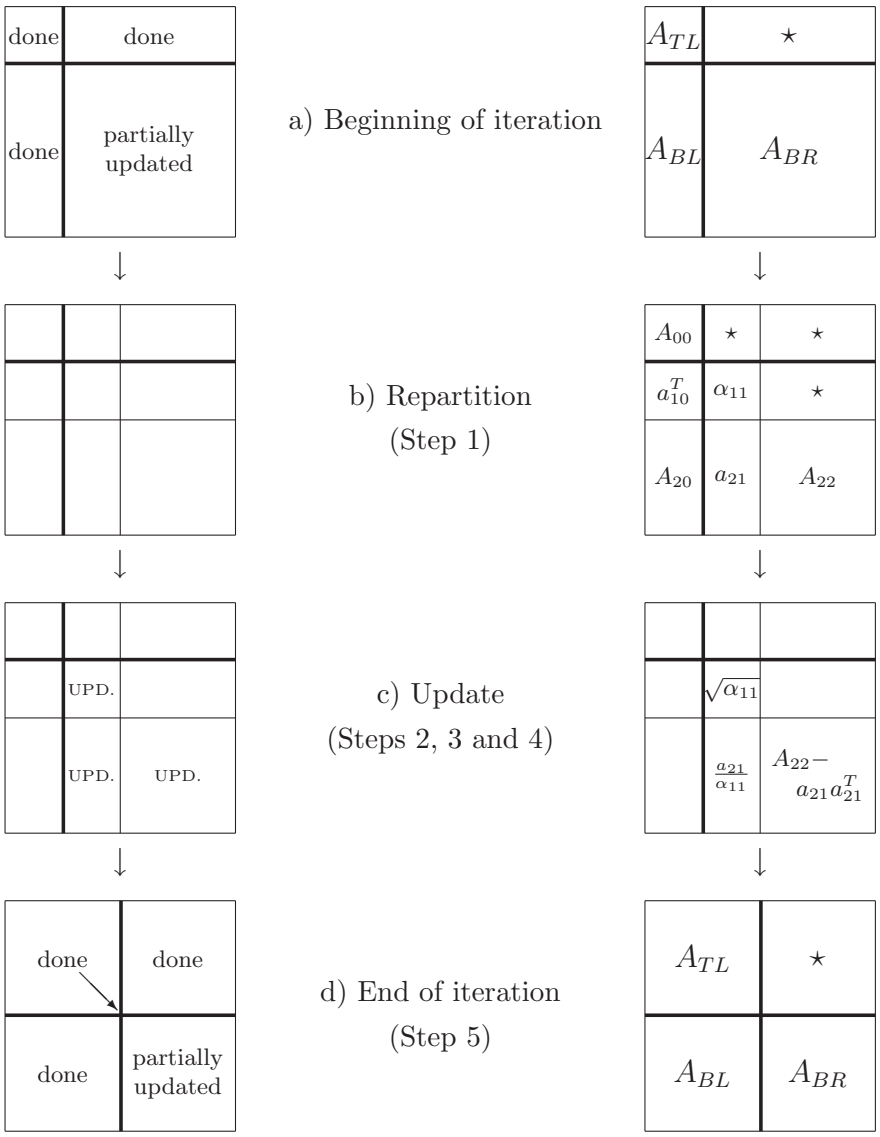

Figure A.1: Left: Progression of pictures that explain the algorithm for computing the Cholesky factorization. Right: Same pictures, annotated with labels and updates.

updated. The algorithm is at the stage in which quadrants A_{TL} and A_{BL} have been computed, i.e., they contain the final result, while quadrant A_{BR} still remains to be computed.

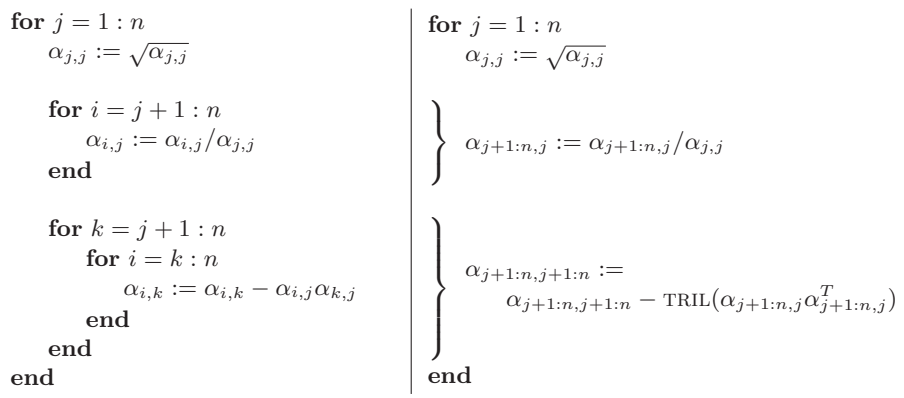

Figure A.2: Formulations of the Cholesky factorization that expose indices. The function $\text{TRIL}(X)$ denotes the lower triangular part of matrix X.

b) Quadrant A_{BR} is partitioned as in Step 1 of the algorithm, and quadrants A_{BL} and A_{TR} are partitioned accordingly. A_{BR} and A_{BL}, respectively, become $\left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right)$ and $\left(\begin{array}{c} a_{10}^T \\ \hline A_{20} \end{array}\right)$.

c) Regions α_{11}, a_{21} and A_{22} are updated performing the computations deduced at Steps 2, 3 and 4.

d) Quadrant A_{TL} is expanded, to keep track of the progress made: it now includes the regions a_{10}^T and α_{11} too. As a consequence, quadrant A_{BR}, which remains to be factored, shrinks down. The other quadrants are updated accordingly.

Algorithm 1 is typically represented using a combination of nested loops and indicating entry by entry the operations performed: see Fig. A.2 (left). In the same figure, on the right, the same algorithm is presented in a more concise way, using the popular "Matlab-like" notation to compact operations on vectors and matrices into a single statement by means of the : operator. In both cases it is difficult to catch the relation between the steps given in Algorithm 1 or Fig. A.1 and the representation in Fig. A.2. The problem lies in the fact that the level of abstraction used to derive and describe the algorithm is different from the level of abstraction

for $j = 1 : n$ **in steps of** n_b
$b := \min(n - j + 1, n_b)$
$A_{j:j+b-1,j:j+b-1} := \Gamma(A_{j:j+b-1,j:j+b-1})$
$A_{j+b:n,j:j+b-1} := A_{j+b:n,j:j+b-1} A^{-T}_{j:j+b-1,j:j+b-1}$
$A_{j+b:n,j+b:n} := A_{j+b:n,j+b:n} - \text{TRIL}(A_{j+b:n,j:j+b-1} A^{T}_{j+b:n,j:j+b-1})$
end

Figure A.3: Blocked algorithm for computing the Cholesky factorization. Here n_b is the block size used by the algorithm.

used to represent it.

The relation between the algorithm and its representation becomes even more obscure when a blocked algorithm is considered (see Fig. A.3). The intricate indexing makes it impossible to create a picture of what parts of the matrix are used and or updated. In contrast, by observing that the entries of matrix A are used in a consistent way, only depending on the region they belong to, it comes natural to try to identify what the different regions are. FLAME employs a notation that exposes the regions of matrices which are handled in the computations [10]. This is accomplished by means of the three constructs **Partition, Repartition** and **Continue with**, as shown in Fig. A.4. On the left and the right of the picture we display, respectively, the unblocked and blocked versions of Algorithm 1 using a high level notation, contrasting the explicit indexing used in Figures A.2 and A.3.

Although the algorithms in Figure A.4 are not as concise as the Matlab-like ones, they exhibit the structure common to both the unblocked and blocked versions[1] and they capture to a large degree the verbal description of the algorithm; we believe that our notation reduces both the effort required to interpret the algorithm and the need for additional explanations. Furthermore, the notation in Figs. A.1 and A.2

[1]The same structure is shared by a family of algorithms computing the same operation (see Chapter 2).

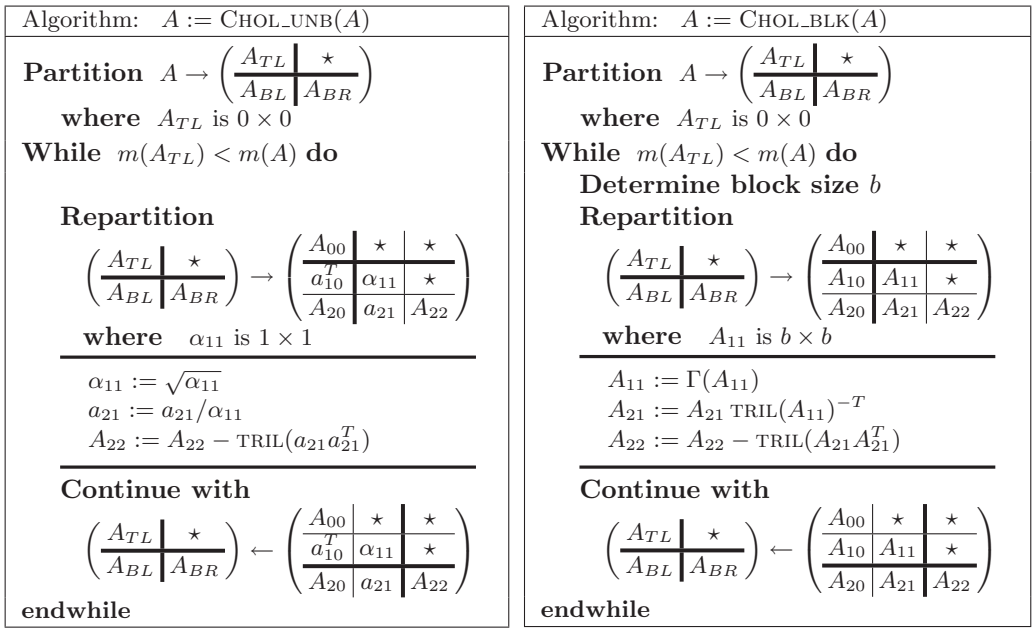

Figure A.4: Unblocked and blocked algorithms for computing the Cholesky factorization.

allows one to easily express the contents of matrix A at the beginning of the iteration:

$$A = \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L_{TL} & \star \\ \hline L_{BL} & \hat{A}_{BR} - \text{TRIL}(L_{BL}L_{BL}^T) \end{array}\right),$$

where $L_{TL} = \Gamma(\hat{A}_{TL})$, $L_{BL} = \hat{A}_{BL}L_{TL}^{-T}$, and $\hat{A}_{TL}, \hat{A}_{BL}$ and $\hat{A}_{BR}$ denote the initial contents of the quadrants A_{TL}, A_{BL} and A_{BR}, respectively. This feature will prove to be fundamental in our research, as it allows us to formally prove the correctness of the algorithms we generate.

A.2 Application Program Interfaces (APIs)

In the numerical dense linear algebra community it is customary to code algorithms by either a Matlab-like entry-by-entry notation (Figs. A.2 and A.3) or by explicitly

invoking BLAS routines (Fig. 1.6). In both cases, it is difficult to relate the code to the algorithm description, and tasks like debugging, code maintenance, and porting are challenging even for the experts.

Our approach is different: in the previous section we identified the level of abstraction at which it is convenient to reason about linear algebra algorithms; now we want use programming APIs to code algorithms at the same level of abstraction.

The goal is achieved by implementing the constructs `FLA_Part*`, `FLA_Repart*`, and `FLA_Cont_with*`, and using an object oriented approach to represent matrices (vectors and scalars are special instances of a matrix) [8]. Figs. 1.5 and 1.7 contain the Matlab and C implementations of Algorithm 1 for computing the blocked Cholesky factorization; notice the resemblance with the algorithm description in Fig. A.4 (right).

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[2] R. H. Bartels and G. W. Stewart. Solution of the matrix equation AX + XB = C. *Commun. ACM*, 15(9):820–826, 1972.

[3] P. Benner, V. Mehrmann, V. Sima, S. Van Huffel, and A. Varga. SLICOT—A subroutine library in systems and control theory. In B.N. Datta, editor, *Applied and Computational Control, Signals, and Circuits*, volume 1, pages 499–539, Boston, 1998. Birkhuser.

[4] Paolo Bientinesi, John A. Gunnels, Fred G. Gustavson, Greg M. Henry, Margaret E. Myers, Enrique S. Quintana-Orti, and Robert A. van de Geijn. Rapid development of high-performance linear algebra libraries. In *Proceedings of PARA'04 State-of-the-Art in Scientific Computing*, June 20-23 2004.

[5] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1), March 2005.

[6] Paolo Bientinesi, Brian Gunter, and Robert van de Geijn. Families of algorithms

related to the inversion of a symmetric positive definite matrix. Technical Report FLAME Working Note 19, TR-06-20, Department of Computer Sciences, The University of Texas at Austin, April 2006.

[7] Paolo Bientinesi, Sergey Kolos, and Robert van de Geijn. Automatic derivation of linear algebra algorithms with application to control theory. In *Proceedings of PARA'04 State-of-the-Art in Scientific Computing*, Copenhagen, Denmark, June 20-23 2004.

[8] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Transactions on Mathematical Software*, 31(1), March 2005.

[9] Paolo Bientinesi and Robert van de Geijn. Formal correctness and stability of dense linear algebra algorithms. In *Proceedings of the 17th IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation*, July 11-16 2005. To appear.

[10] Paolo Bientinesi and Robert A. van de Geijn. Representing dense linear algebra algorithms: A farewell to indices. Technical Report FLAME Working Note 17, TR-06-10, Department of Computer Sciences, The University of Texas at Austin, February 2006.

[11] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[12] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley.

ScaLAPACK user's guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[13] Edsger W. Dijkstra. The humble programmer. (EWD340). *Commun. ACM*, 15(10):859–866, 1972. Turing Award lecture.

[14] Edsger W. Dijkstra. Programming as a discipline of mathematical nature. (EWD361). *Am. Math. Monthly*, 81(6):608–612, 1974.

[15] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[16] Jack Dongarra. The LINPACK benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474, London, UK, 1988. Springer-Verlag.

[17] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[18] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[19] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[20] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3nd edition, 1996.

[21] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*. submitted.

[22] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer Verlag, 1992.

[23] John A. Gunnels. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms.* PhD thesis, Department of Computer Sciences, The University of Texas, December 2001.

[24] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[25] R. J. Hanson, F. T. Krogh, and C. L. Lawson. Improving the efficiency of portable software for linear algebra. *SIGNUM Newsl.*, 8(4):16–16, 1973.

[26] Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[27] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969.

[29] Isak Jonsson and Bo Kågström. Recursive blocked algorithms for solving triangular systems—part i: one-sided and coupled sylvester-type matrix equations. *ACM Transactions on Mathematical Software*, 28(4):392–415, 2002.

[30] Isak Jonsson and Bo Kågström. Recursive blocked algorithms for solving triangular systems—part ii: Two-sided and generalized sylvester and lyapunov matrix equations. *ACM Transactions on Mathematical Software*, 28(4):416–435, 2002.

[31] Bo Kågström and Peter Poromaa. LAPACK-style algorithms and software for solving the generalized Sylvester equation and estimating the separation between regular matrix pairs. *ACM Transactions on Mathematical Software*, 22(1):78–103, 1996.

[32] LAPACK – Linear Algebra PACKage. http://www.netlib.org/lapack/.

[33] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[34] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *TOMS*, 29(2):218–243, June 2003.

[35] B. T. Smith et al. *Matrix Eigensystem Routines – EISPACK Guide*. Lecture Notes in Computer Science 6. Springer-Verlag, New York, second edition, 1976.

[36] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[37] R. Clint Whaley. *Automated Empirical Optimization of High Performance Floating Point Kernels*. PhD thesis, Department of Computer Sciences, The Florida State University, Fall 2004.

[38] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings.

[39] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Notes on Applied Science No. 32, Her Majesty's Stationery Office, London, 1963. Also published by Prentice-Hall, Englewood Cliffs, NJ, USA. Reprinted by Dover, New York, 1994.

[40] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford University Press, Oxford, England, 1965.

[41] Stephen Wolfram. *The Mathematica Book: 3rd Edition.* Cambridge University Press, 1996.

[42] Kamen Yotov, Xiaoming Li, Gang Ren, Maria Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

Vita

Paolo Bientinesi was born in Livorno, Italy, on October 27, 1973, son of Demo Bientinesi and Emilia Balducci; he is ten years younger than his sister Paola. He grew up in Castiglioncello, on the Tuscan coast. After completing his high-school work at the Liceo Scientifico Enrico Fermi of Cecina in 1992, he attended the University of Pisa. In April 1998 he received a *Summa cum Laude* Laurea degree in Computer Science. Thereafter, Paolo served as officer in the Italian Navy as head of a class of 170 cadets. In January 2000 he moved to the US where he spent a semester at the Texas Institute for Computational and Applied Mathematics. In August 2000 he joined the PhD program in Computer Sciences of the University of Texas at Austin.

Permanent Address: via Aurelia 570
57012 Castiglioncello (LI)
ITALY

This dissertation was typeset with LaTeX 2_ε[2] by the author.

[2] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.