Graz University of Technology

Doctoral Thesis

Efficient Floating-Point Implementation of Signal Processing Algorithms on Reconfigurable Hardware

Thang Viet Huynh

Signal Processing and Speech Communication Laboratory
Graz University of Technology, Austria

Supervisors:
Prof. DI Dr. Gernot Kubin, Graz University of Technology, Austria
DI Dr. Manfred Mücke, University of Vienna, Austria

Examiners:
Prof. DI Dr. Gernot Kubin, Graz University of Technology, Austria
Prof. Markus Püschel, ETH Zürich, Switzerland

Graz, July 9, 2012

Abstract

This doctoral thesis aims at optimising the floating-point implementations of signal processing algorithms on reconfigurable hardware with respect to accuracy, hardware resource and execution time. It is known that reduced precision in floating-point arithmetic operations on reconfigurable hardware directly translates into increased parallelism and peak performance. As a result, efficient implementations can be obtained by choosing the minimal acceptable precision for floating-point operations. Furthermore, custom-precision floating-point operations allow for trading accuracy with parallelism and performance. We use Affine Arithmetic (AA) for modeling the rounding errors of floating-point computations. The derived rounding error bound by the AA-based error model is then used to determine the smallest mantissa bit width of custom-precision floating-point number formats needed for guaranteeing the desired accuracy of floating-point applications.

In this work, we implement the first Matlab-based framework for performing rounding error analysis and numerical range evaluation of arbitrary floating-point algorithms using the AA-based error model. Our framework enables users to best reuse their own existing Matlab code to effectively conduct rounding error analysis tasks and run bit-true custom-precision computations of floating-point algorithms in Matlab for verification.

We apply the AA-based error analysis technique and our Matlab-based framework to the floating-point rounding error evaluation and optimal uniform bit width allocation of two signal and speech processing applications: *i)* the floating-point dot-product and *ii)* the iterative Levinson-Durbin algorithm for linear prediction and autoregressive modeling. For the floating-point dot-product, it is shown that the AA-based error model can provide tighter rounding error bounds compared to existing error analysis techniques. This corresponds to the overestimation of up to 2 mantissa bits compared to those estimated by running extensive simulations. For the iterative Levinson-Durbin algorithm, the AA-based error analysis technique can model accurately the rounding errors of the coefficients when using a restricted range for the input parameters. When using a general range for the input parameters, the AA-based error analysis technique can give a qualitative estimate for the error bound of the coefficients.

Kurzfassung

Ziel dieser Dissertation ist die Optimierung von Gleitkomma-Implementierungen von Algorithmen in der Signalverarbeitung auf rekonfigurierbarer Hardware in Bezug auf Genauigkeit, Hardware-Ressourcen und Ausführungszeit. Es ist bekannt, dass eine reduzierte Genauigkeit in Gleitkomma-Rechenoperationen auf rekonfigurierbarer Hardware direkt in eine höhere Parallelität und Spitzenleistung resultiert. Eine effiziente Implementierung kann durch Verwendung minimaler, akzeptabler Genauigkeit von Gleitkomma-Operationen erreicht werden. Wir verwenden Affine Arithmetik (AA) für die Modellierung der Rundungsfehler in Gleitkomma-Berechnungen. Die hergeleitete Schranke für Rundungsfehler wird dann verwendet, um die kleinste Bit-breite für die Mantisse der angepassten Gleitkomma-Zahlen-Formate zu bestimmen. Diese gewährleistet die gewünschte Genauigkeit der Gleitkomma-Anwendung.

Außerdem implementieren wir das erste Matlab-basierte Framework für die Durchführung einer Rundungsfehleranalyse sowie die Auswertung des numerischen Bereiches von beliebigen Gleitkomma-Algorithmen mit Hilfe des AA-basierten Fehlermodells. Unser Framework ermöglicht es Rundungsfehler und Bit-genaue Berechnungen von Gleitkomma-Rechenoperationen mit speziell angepasster Genauigkeit zu berechnen und mit vorhandenem Matlab-Code zu evaluieren.

Wir wenden die AA-basierte Fehleranalyse-Technik und das Matlab-basierte Framework auf zwei Anwendungen in der Signal- und Sprachverarbeitung an: *i)* das Gleitkomma-Skalarprodukt und *ii)* der iterative Levinson-Durbin-Algorithmus. Für das Gleitkomma-Skalarprodukt wird gezeigt, dass der AA-basierte Rundungs-Operator engere Schranken für den Rundungsfehler liefert als bestehende Techniken zur Fehleranalyse. Dies entspricht einer Überschätzung von bis zu 2 Bits für die Mantisse im Gegensatz zur geschätzten Bit-breite aus umfangreichen Simulationen. Im iterativen Levinson-Durbin-Algorithmus kann die AA-basierte Fehleranalyse-Technik Rundungsfehler der Koeffizienten genau modellieren, indem ein eingeschränkter Bereich für die Eingangsparameter verwendet wird. Verwendet man hingegen einen allgemeinen Bereich für die Eingangsparameter, gibt die AA-basierte Fehleranalyse-Technik eine qualitative Abschätzung für die Fehlerschranken der Koeffizienten an.

Contents

List of Figures

List of Tables

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used resources.

....	....
(date)	(signature)

Acknowledgement

It is my great pleasure to acknowledge all the support I received during my PhD studies.

First of all, I would greatly appreciate my supervisor, Prof. Gernot Kubin, for his excellent guidance during my doctoral studies. This thesis cannot be written without his support. When working with him, I truly admire his ability of having very simple and logical explanation. Thank you for supporting me over the past years!

I am greatly thankful to my co-supervisor, Dr. Manfred Mücke, for his great enthusiasm when guiding me during my PhD studying time. I am very happy to work with him. Thank you for your countless discussions, hints and supports making this thesis be written!

I am very grateful to Prof. Markus Püschel for the interesting discussions and valuable comments he gave that helped to improve the overall quality of this thesis. It is my great honor to have my thesis examined by him.

Many thanks to all my colleagues at Signal Processing and Speech Communication Laboratory (SPSC) who shared "SPSC Kaffee" with me and helped me during my doctoral studies in Graz. My special thanks to our administrative and IT staffs - Karin Karner, Markus Köberl and Andreas Läβer - for their wonderful supports to make all the administrative work and Linux programs run smoothly. Thanks to Wolfgang Jaeger for his indispensable and patient work in setting up the Stretch S6 PCIe board. Thanks to Bernd Lesser and Anna Katharina Fuchs for sharing the office, having very interesting discussions, and helping me in my daily life.

I would express my gratefulness to Dr. Wilfried N. Gansterer, Head of The Research Lab Computational Technologies and Applications (RLCTA), University of Vienna, for his valuable supports during my doctoral studies.

A very special thank to Ms. Astrid Brodtrager at the Dean's office of the Faculty of Electrical and Information Engineering, TU Graz for her great help in the submission of this thesis.

I would very much thank the Austrian Academic Exchange Service (ÖAD) for giving me the chance to pursue my PhD studies in Austria. My special thanks to Mag. Christina Düss at the ÖAD Regional Office Graz for helping me during the past years.

I am grateful to my colleagues at the Department of Electronic-Telecommunication Engineering at Danang University of Technology, Vietnam, for their great help when I was studying abroad. I would also like to take this chance to express my gratefulness to Dr. Pham Van Tuan who encouraged me to pursue my doctoral studies in SPSC.

Many thanks to all of my Vietnamese friends in Austria and particularly in Graz, who have experienced the students' lives with me.

I would like to thank my family for their understandings and supports during the time I was abroad.

Last but not least, I deeply thank my beloved wife, Nguyen Thi Thuy Phuong, who always stayed by my side, motivated and supported me while writing this thesis.

Thang Viet Huynh
Graz, July 9, 2012.

1 Introduction

The use of floating-point arithmetic has increased dramatically in digital signal processing (DSP) applications over the last two decades due to the rapid development of hardware technology [1]. Dating back to 1973, the first floating-point digital filter was built by Lacroix in TTL (Transitor-Transitor Logic) circuits (see [2] and the references therein). Using floating-point arithmetic for DSP applications has several benefits compared to using traditionally fixed-point arithmetic. One of those benefits is the larger dynamic range of floating-point numbers, generally leading to a more accurate computed result. The design and implementation of DSP algorithms using floating-point arithmetic are, however, much more complex than using fixed-point arithmetic, making it hard to achieve efficient floating-point implementations of signal processing algorithms.

This doctoral thesis aims at obtaining efficient implementations of floating-point signal processing algorithms on reconfigurable hardware with respect to accuracy, hardware resources and execution time.

For achieving this goal, this thesis studies the relation between the desired accuracy and the precision of floating-point arithmetic operations in order to identify the smallest bit width of floating-point operands necessary for guaranteeing the originally desired accuracy.

Going a step further, this work exploits custom-precision floating-point operations on reconfigurable hardware and investigates the relation between the desired accuracy and the resulting hardware resources and sustained performance of custom-precision floating-point signal processing algorithms implemented on reconfigurable fabrics.

1.1 Motivation

The motivation for conducting this doctoral thesis first comes from the development of new reconfigurable hardware architectures that allow for more flexible customisation of the data-path while still offering the same relative performance improvement as it was offered by traditional reconfigurable platforms, yet at a lower power consumption and lower cost. In this thesis, we investigate a hybrid reconfigurable CPU architecture.

Novel methods for modeling and evaluating finite-precision arithmetic operations have recently emerged, like using interval arithmetic (IA) [3] or affine arithmetic (AA) [4] for floating-point error analysis, allowing for more accurate (or tighter) estimates of the rounding error bound of floating-point algorithms in comparison with the rounding error bound derived by using the conventional floating-point error model [5]. This is extremely useful for bit width allocation for hardware implementation on reconfigurable architectures as it can provide more realistic bit width estimates, and, as a result, lead to more efficient hardware implementation. In this thesis, we explore AA for the floating-point rounding error analysis and bit width allocation.

Autoregressive modeling and linear prediction [6–8] are two very important and popular applications in signal and speech processing. These two applications involve solving a Toeplitz system of linear equations, for which the iterative Levinson-Durbin algorithm is the most ubiquitous solver. Efficient implementations of the Levinson-Durbin algorithm using floating-point arithmetic have not yet been studied on reconfigurable hardware. One of the main reasons is that existing floating-point rounding error bounds derived for the Levinson-Durbin algorithm by conventional error modeling methods are too pessimistic compared to the real errors observed in practice, rendering those bounds unsuitable for bit width allocation. We will investigate if AA for modeling and evaluating floating-point arithmetic operations can help derive a tighter rounding error bound for the iterative Levinson-Durbin algorithm and therefore give a better bit width estimate for floating-point hardware implementation. Next, given acceptable bit width estimates for the floating-point Levinson-Durbin algorithm derived by the new method, we would like to know the resulting performance and hardware resource requirements.

1.2 Related Work

This work relates to several research areas including reconfigurable computing, rounding error modeling and optimal bit width allocation, custom-precision floating-point arithmetic, and signal and speech processing. There exist many related works in the literature, among which we selectively summarize in this section the most relevant papers and doctoral theses in order to differentiate this work from preceding works. Note that related work will also be presented in relevant parts of the thesis.

A survey on architectures and design methods in reconfigurable computing is presented in [9]. Hybrid reconfigurable CPUs can be seen as a combination of two technologies: CPU instruction set architecture and reconfigurable logics which allow for the customisation of the data-path through extension instructions. An overview of hybrid reconfigurable CPUs is presented in [10, 11]. It is known that reduced precision in floating-point arithmetic operations typically brings performance gain on fixed data-path architectures and both performance and parallelism gains on reconfigurable architectures [12, 13]. Several hybrid reconfigurable CPU architectures exist [14–16]. This thesis aims to verify this assumption on hybrid reconfigurable CPUs. The work of Huynh [17] is the first one to study the customisation of the instruction set of hybrid reconfigurable CPUs for multi-tasking embedded systems, in which, similar to our work, a Stretch reconfigurable CPU is used. However, the work of Huynh focuses on dynamic instruction set customisation and runtime reconfiguration for hybrid reconfigurable CPUs and considers only fixed-point or integer arithmetic implementations of user-defined functional units. Custom-precision floating-point arithmetic on hybrid reconfigurable CPUs is still unexplored.

One challenge of custom-precision floating-point applications on reconfigurable hardware is to identify the smallest precision of floating-point operands. This involves floating-point rounding error modeling and bit width allocation. Different methods can be used: extensive simulations [18], conventional error analysis [5], automatic differentiation [19, 20], perturbation analysis [21], IA [3, 22], and AA [4, 23–26]. A summary of methods and accompanying software tools for rounding error modeling and bit width allocation can be found in [9, 24].

In this thesis, we use AA to model the rounding error of floating-point algorithms. Therefore, we are more interested in related work using AA in floating-point applications. The work of Fang [4, 23] is the first to apply AA [27] for modeling the rounding error of floating-point arithmetic for linear transforms. Going a step further, a probabilistic enhancement to AA for floating-point rounding error estimation has been developed [28], potentially allowing for a more accurately estimated interval. The core is to associate a probabilistic bounding operator with AA intervals. Fang suggests AA-based rounding error models for floating-point number and operations and applies those models to rounding error analysis of DSP linear kernels including Finite Impulse Response (FIR) filters and the Discrete Cosine Transform (DCT). However, the AA-based rounding error analysis of complex algorithms consisting of non-linear operations and iterative computational sequences are still unexplored.

Cybenko [29, 30] performs finite-precision rounding error analyses for some signal processing algorithms using the conventional rounding error model. Floating-point and fixed-point error bounds for the Levinson-Durbin algorithm are, for the first time, presented by Cybenko [30]. Those bounds are however very conservative compared to the rounding errors observed in speech processing applications making them unsuitable for optimal bit width allocation for hardware implementation of the Levinson-Durbin algorithm. Existing implementations of the Levinson-Durbin algorithm can be found in [31–35]. Note that other Toeplitz solvers are summarized later in Section 5.3.

1.3 Contributions

The scientific contributions of this thesis are:

❑ **C1**. This thesis is the first to investigate the performance and resource usage of floating-point arithmetic operations as a function of precisions (lower than double-precision) on a *hybrid* reconfigurable CPU.

❑ **C2**. This thesis describes the implementation of the first Matlab-based tool, the `AAFloat` class, allowing for efficient and reliable error modeling and estimation of floating-point algorithms using AA, performance speed-up and flexible handling of exceptional cases. In addition, an AA-based error model for a floating-point fused multiply-add operation is, for the first time, suggested in this work.

❑ **C3**. This thesis shows more realistic AA-based error bounds for floating-point dot-products compared to the bounds estimated by using conventional error analysis techniques. Going a step further, this thesis derives an analytical error model for the floating-point dot-product as a function of vector length, precision, numerical range and dot-product implementation variant, allowing for efficient bit width allocation and design space exploration.

❑ **C4**. This thesis is the first to apply AA for the evaluation of the rounding error propagation in the iterative Levinson-Durbin algorithm. It is also the first time that a custom-precision floating-point Levinson-Durbin implementation in Matlab is presented and evaluated with respect to the rounding error. Furthermore, this thesis suggests to incorporate the additional knowledge on the Levinson-Durbin algorithm from analytical work with the AA-based model, via applying an enforced bound on the range, in order to alleviate the overestimation effect, thereby obtaining more sensible qualitative estimates for the error bounds.

1.4 Publications

The following publications have been written during this doctoral work (in chronological order):

- T. V. Huynh and M. Mücke, *Exploiting Reconfigurable Hardware to Provide Native Support of Double-Precision Arithmetic on Embedded CPUs*, in Research Poster Session, International Supercomputing Conference (ISC), Hamburg, Germany, 2010 [36]
- T. V. Huynh, M. Mücke, and W. N. Gansterer, *Native Double-Precision LINPACK Implementation on a Hybrid Reconfigurable CPU*, in 18th Reconfigurable Architectures Workshop (RAW), IEEE, Anchorage, Alaska, USA, May 2011 [37]
- T. V. Huynh and M. Mücke, *Error Analysis and Precision Estimation for Floating-Point Dot-Products using Affine Arithmetic*, in The 2011 International Conference on Advanced Technology for Communications (ATC'2011). IEEE, Danang, Vietnam, Aug. 2011 [38]
- T. V. Huynh, M. Mücke, and W. N. Gansterer, *Evaluation of the Stretch S6 Hybrid Reconfigurable Embedded CPU Architecture for Power-Efficient Scientific Computing*, in International Conference on Computational Science (ICCS 2012), Elsevier, Omaha, Nebraska, USA, Jun 2012 [39]
- T. V. Huynh and M. Mücke, *A Tool for Floating-Point Rounding Error Modeling and Estimation in Scientific Computing Applications Using Affine Arithmetic*, in preparation for submission to ACM Transactions on Mathematical Software (TOMS) [40]

for which publications [36, 37, 39] correspond to scientific contribution **C1**, publication [38] is part of scientific contribution **C3**, and publication [40] will cover scientific contribution **C2**.

1.5 Thesis Outline

The rest of the thesis is organized as follows:

Chapter 2 presents the performance of floating-point operations as a function of precision on different hardware architectures. Section 2.3 specifically focuses on the Stretch S6 hybrid reconfigurable CPU and presents the area performance and throughput performance offered by reduced-precision floating-point arithmetic operations. The *precision-to-performance* relation presented in Chapter 2 is the motivation for custom-precision floating-point implementations of signal processing algorithms on reconfigurable hardware and for rounding error analysis and optimal bit width allocation of floating-point algorithms that will be studied later in Chapter 3, Chapter 4, and Chapter 5.

Chapter 3 presents the fundamentals of floating-point error analysis using affine arithmetic which is the basis for the implementation of a Matlab-based framework for the automation of floating-point rounding error analysis of arbitrary floating-point algorithms described later in Chapter 4. In Chapter 3, we also propose an affine error model for a floating-point fused multiply-accumulate operation.

Chapter 4 presents our implementation of a Matlab-based framework for rounding error and numerical range evaluation of floating-point algorithms using affine arithmetic error modeling. We first give an overview of the framework and then describe in more detail the usage of the `AAFloat` class implemented in Matlab as well as our implementation of custom-precision floating-point arithmetic operations for Matlab via the MPFR [41] library. The chapter presents demonstrative examples showing how to use our Matlab tool for efficient error estimation.

Chapter 5 applies the AA-based error model and Matlab-based tool to two applications. In the first application, we estimate the rounding errors of different floating-point dot-product implementation variants. In the second application, we perform floating-point rounding error analysis for the iterative Levinson-Durbin algorithm. The rounding error bounds of floating-point dot-products can be used to identify the optimal mantissa bit width for hardware implementations of FIR filters in signal processing applications. The rounding error bounds of the Levinson-Durbin algorithm in the second application are necessary for floating-point implementations of linear prediction in speech processing and autoregressive modeling in signal processing.

Finally, Chapter 6 summarizes the contributions of the thesis and discusses potential future research directions.

2 Floating-Point Arithmetic Performance

2.1 Introduction

The significant benefit on area and performance offered by reduced-precision floating-point arithmetic operations is the motivation for custom-precision and mixed-precision floating-point implementations of signal processing and scientific computing applications.

This chapter presents the floating-point operation performance as a function of precision on different hardware architectures. It is known that reduced precision in floating-point arithmetic operations directly translates into increased parallelism and peak performance on reconfigurable fabrics, e.g., presented in [13]. In this chapter, we will specifically illustrate this relation on the Stretch S6 hybrid reconfigurable CPU. Given a specific precision, the corresponding performance and area of floating-point operations can be estimated.

The challenge of implementing custom-precision or mixed-precision floating-point applications on reconfigurable hardware is to identify the minimum precision of floating-point operands for achieving the highest application performance while guaranteeing the required accuracy of the respective applications. To solve the challenging problem of optimal precision estimation, a relation between the user's required accuracy of the floating-point algorithmic final result and the working precision needs to be established. That task is called floating-point error analysis and will be presented later in Chapters 3, 4 and 5. We would like to emphasize that the performance of floating-point operations on different hardware architectures as a function of the precision, presented in this chapter, is the motivation for studies and discussions on floating-point error analysis and bit width allocation presented later in the other chapters. Therefore, we believe this chapter will be interesting for a relatively wide range of readers, i.e., not only for readers in the hardware design area but also for the ones coming from the signal processing and scientific computing areas as well.

We will first give a short summary of floating-point performance on general purpose CPUs, graphics processing units (GPUs) and field programmable gate arrays (FPGAs). Next, we focus on investigating the area and performance achievable on the Stretch S6 hybrid reconfigurable CPU using custom-precision floating-point arithmetic. Using the LINPACK benchmark, we specifically show the impact of precisions lower than double-precision on the throughput of floating-point applications. The chapter closes with some concluding remarks and highlights the motivation for the next chapters.

Table 2.1 Summary of SP and DP floating-point performance on CPU, GPU and FPGA

	CPU Intel i7-965	GPU NVIDIA Tesla C1060	FPGA Virtex-6 SX475T
Frequency (GHz)	3.2	1.3	< 0.55
Peak SP performance (GFlop/s)	102.4	936	550
DP:SP performance ratio	**1:2**	**1:12**	$\approx$ **1:4**

2.2 Floating-Point Performance on CPUs, GPUs and FPGAs

Heterogeneous computing, i.e., using different types of processing units for maximising performance, has become attractive during the last decade since it offers high peak performance and power- and/or cost-effective designs compared to using traditional CPUs. The current trend for increasing performance is to make use of parallelism via the combinations of CPU-with-GPU or CPU-with-FPGA instead of increasing clock frequency. An extensive discussion on state-of-the-art heterogeneous architectures is presented in [12].

A GPU is a symmetric multi-core processor that is exclusive accessed and controlled by the CPU via the PCI express bus. Traditionally, the GPU was designed for use in image processing to render geometric objects. Recent GPUs are more general and have been widely used for high performance computing. The three major GPU vendors are NVIDIA, AMD and Intel.

An FPGA is a set of configurable logic blocks, digital signal processing blocks, and optional traditional CPU cores that are all connected via a reconfigurable interconnect. When configured, FPGAs function like user-defined application-specific integrated circuits (ASICs). In a heterogeneous system, a FPGA is often connected with the traditional CPU via the high-speed HyperTransport bus.

We focus on floating-point performance of each of the above processing units, i.e., CPU, GPU and FPGA. Table 2.1, extracted from [12], summaries the single-precision (SP) and double-precision (DP) floating-point performance of some recent CPU, GPU and FPGA architectures. The CPU is an Intel Core i7-965 Quad Extreme, the NVIDIA GPU is the Tesla C1060, and the FPGA is the Virtex-6 SX475T. The numbers in Table 2.1 are reported per physical chip.

The GPU gives the best peak performance when it comes with single-precision floating-point arithmetic, offering an order of magnitude better performance compared to others. Instead of using double precision, exploiting single-precision halves the storage and bandwidth requirements and increases the peak performance significantly from two up to twelve times (i.e., on GPUs) depending on the architecture used (see Table 2.1). More specifically, the CPU offers a linear performance improvement with decreasing precision, while the FPGA provides quadratic performance improvement when decreasing precision (Table 2.1).

Using high precision is not always necessary for some applications, as the desired accuracy of an algorithm might be less than double precision and even less than single precision. Some real world signal processing applications can actually tolerate some computational error in the intermediate or even final results [22, 42]. On FPGAs, using the lowest possible precision (or custom-precision) can give significant benefits in storage and bandwidth requirements, as well as in parallelism, thereby further resulting in increased performance. Figure 2.1, adopted from [13], demonstrates the peak performance of a binary-tree based custom-precision floating-point dot-product as a function of precision (i.e., the x-axis in Figure 2.1) on an Altera Cyclone II EP2C70 FPGA. The lower part of Figure 2.1

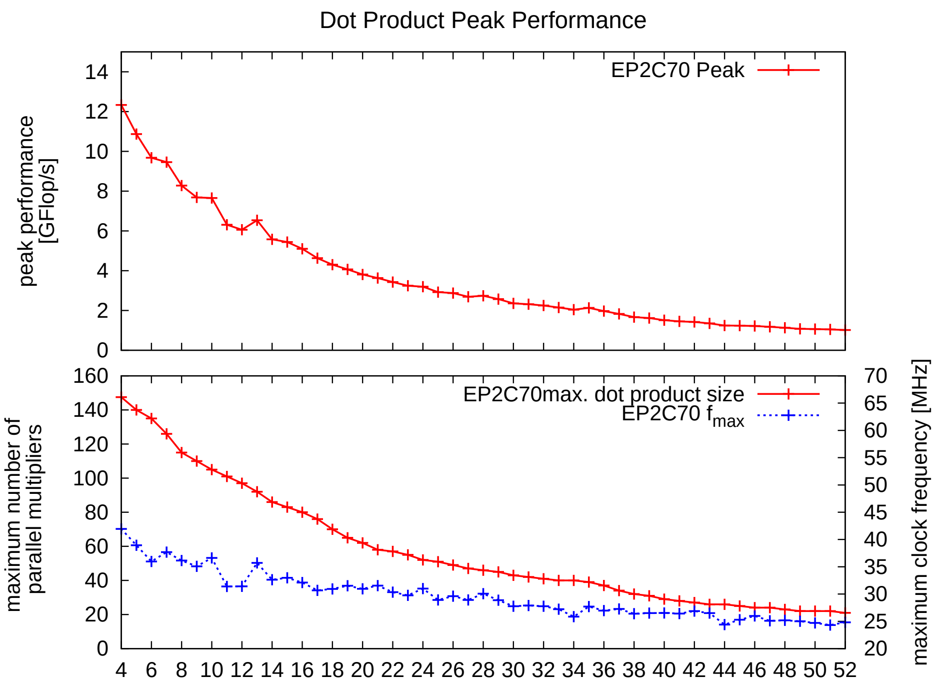


Figure 2.1 Peak performance of binary-tree based custom-precision floating-point dot-product on FPGAs versus precision

shows superlinear gains in the number of parallel multipliers (red curve, corresponding to left y-axis) implementable on the FPGA and respective achievable clock frequency (blue curve, corresponding to right y-axis) while reducing the precision. As a consequence of these increases, the peak performance scales superlinearly with decreased mantissa bit width (red curve in upper part of Figure 2.1).

2.3 Floating-Point Performance on Hybrid Reconfigurable CPUs

Hybrid reconfigurable CPUs mix two technologies: CPU instruction set architecture and reconfigurable logics which, in their respective domain, have evolved together with respective tools (compilers and synthesizers) and can typically deliver state-of-the-art performance. As a consequence of this combination, hybrid reconfigurable CPUs allow for customisation of the data-path through extension instructions to improve a given application's performance with minimal development overhead by relying for most parts on a proven static CPU architecture. The more complex the custom instruction, i.e., the higher the equivalent number of instructions from the original instruction set, the higher is typically the achievable performance gain.

Several hybrid reconfigurable CPU architectures exist, among them MOLEN [14], GARP [15] and Stretch [16]. An overview can be found in [10, 11]. An additional issue faced by designers of hybrid reconfigurable CPUs, however, is the question of how to combine best the respective strengths of fixed instruction-set architectures and reconfigurable logic. Reconfigurable fabrics are added to enhance the capabilities of static architectures. While the *peak* performance of hybrid reconfigurable CPUs (subject to application requirements and suitable configuration) is mostly defined by the reconfigurable fabric, the achievable *sustained* performance relies heavily on the (static) interface between reconfigurable fabric and fixed CPU.

Over the last decade, short-vector single-instruction multiple-data (SIMD) units have been integrated into mainstream CPUs [43–45]. This trend has already extended to floating-point units (FPUs). While static FPUs typically provide a linear improvement in parallelism with decreasing precision, FPGAs provide quadratic improvement in parallelism with decreasing precision for selected operations [46, 47]. However, implementation of complex numerical algorithms in FPGAs requiring specification in a hardware description language (HDL) appears both laborious and error-prone. Hybrid reconfigurable CPUs provide a means for efficient coding using a reliable software stack while potentially delivering a superlinear performance gain for lower-precision arithmetic operations. Most publications presenting a reconfigurable CPU include some design space exploration for applications originating from multimedia benchmarks (a favourite usage scenario for reconfigurable CPUs). We believe floating-point applications are underrepresented in design space exploration for hybrid reconfigurable CPUs, potentially leading to suboptimal interface design.

We aim at verifying the assumption of superlinear area- and performance gains for lower-precision arithmetic operations on the Stretch S6 hybrid reconfigurable CPU [16]. This is achieved by running the LINPACK benchmark on the S6 CPU using either double- or single-precision number formats and investigating in detail the interface between reconfigurable fabric and fixed CPU considering typical requirements of extension instructions using single- and double-precision operands.

2.3.1 The Stretch S6 CPU

The Stretch S6 [16] is a hybrid reconfigurable embedded CPU which combines a fixed Tensilica Xtensa LX instruction set architecture with a reconfigurable Stretch extension unit. In the following we describe the S6's architecture and summarize typical application development on the S6 hybrid reconfigurable CPU.

Architecture

Figure 2.2 gives an overview of the Stretch S6 architecture. The 32-bit Xtensa LX core (blue) can run at a clock frequency of up to 300 MHz and the programmable Instruction Set Extension Fabric (ISEF, yellow) can run at clock frequencies identical, 1/2 or 1/3 of the Xtensa clock frequency. The Xtensa core is equipped with an FPU providing native support of the IEEE-754 single precision floating-point arithmetic. Double precision arithmetic, however, has to be emulated in software. The emulation is based on the gcc soft float routines (contained in `libgcc` and normally used by gcc when generation of floating-point instructions is disabled).

ISEF. The ISEF is an array of reconfigurable computational resources, memories, registers and respective interconnect, which can be used to implement user-defined extension instructions. Between the Xtensa core and the ISEF, data is transferred via 128-bit Wide Registers (WR), using a maximum of three registers for input and two registers for output. The ISEF supports full pipelining of extension instructions with up to 27 pipeline stages. The ISEF's computational resources comprise 4096 arithmetic units (AUs) for bitwise addition/subtraction and logic operations and 8192 multiply units (MUs) for bitwise multiply and shift operations. The ISEF features 64KB of embedded RAM (IRAM) which can be accessed from the Xtensa core via fast direct memory access (DMA).

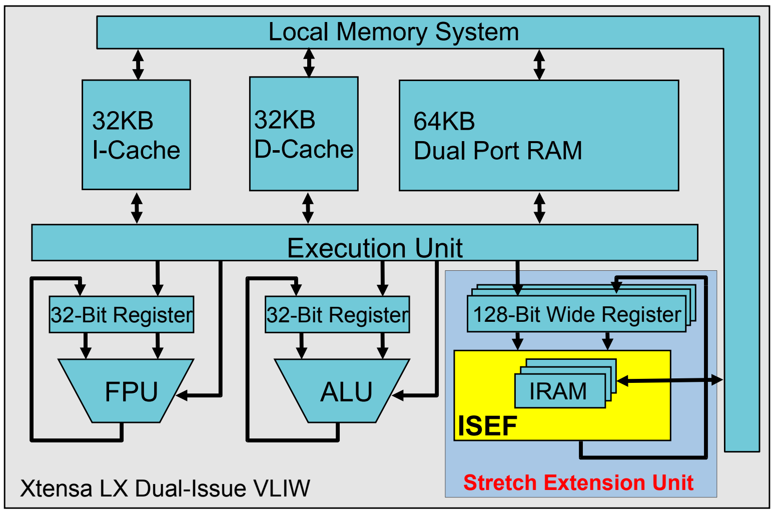


Figure 2.2 Stretch S6 Architecture

There are four fundamental sources of potential performance gains when offloading computations to the ISEF [48]:

- Instruction specialization: As extension instructions serve only a single application, they can be much more specific than general-purpose instructions.
- Spatial parallelism: The ISEF allows for implementation of parallel data paths, limited only by the number of available ISEF resources and the width of input- and output registers.
- Temporal parallelism: Up to 27 pipelining stages.
- Embedded memory: The ISEF features multiple embedded memories providing massive bandwidth at very low latency to access look-up tables or to keep temporary data.

Application Development

Application development for the Stretch CPU typically starts with a new or existing C or C++ program running on a sequential CPU platform [16, 48]. The code is profiled and analysed to identify the code segments, typically inner loops, which consume most of the execution time. These identified code segments will then be replaced by user-defined extension instructions, implemented in the ISEF, and invoked from the main program as C intrinsics.

Source code for an application using extension instructions on a Stretch hybrid CPU is composed of two parts: (*i*) ordinary (ANSI) C/C++ code to implement the application executed on the Xtensa, and (*ii*) Stretch C code to define the extension instructions. Stretch C is ANSI C with a few enhancements and limitations [48]. The enhancements include data types of parameterisable bit width and operators for packing and unpacking bits within longer words. The Stretch C Compiler (SCC), which is based on gcc, maps ordinary C code into a series of instructions to run on the Xtensa processor, and Stretch C code into a bitstream for ISEF configuration. Once the user-defined extension instructions have been defined in Stretch C, the extension instructions are compiled by SCCS. A header file defining the intrinsics associated with the extension instructions is created and included in all ordinary C files in which the extension instructions are used.

The wide registers (WRs) build the interface between ISEF and Xtensa, holding the input to extension instructions as well as the computed result. The Stretch S6 CPU provides a variety of load/store instructions and *byte-streaming channels* for transferring data between memory, Xtensa core and WRs. A typical flow for using a user-defined extension instruction consists of three steps: 1) load data from memory to corresponding WRs, 2) execute the extension instruction as C intrinsic, 3) store the result from WRs to memory or transfer to other Xtensa registers.

Byte-streaming channels. The Stretch provides byte-streaming load-store instructions, which allow for transferring of 1 to 16 bytes between WRs and memory while implicitly updating the memory address with an increment or decrement. The S6 CPU provides three independent load-streams (RAM to WR) and one store-stream (WR to RAM). After initialization, streaming loads and stores take just one cycle to execute, as long as the data resides in on-chip memories, i.e., D-Cache or Dual-Port RAM. The following Stretch-C code example demonstrates byte-streaming to transfer data between wide registers and memory on the Stretch S6 CPU.

```
WR  A1, A2, B;
double x1[VECTOR_LENGTH], x2[VECTOR_LENGTH], y[VECTOR_LENGTH];
double *pxr1, *pxr2, *pyw;
pxr1 = x1; pxr2 = x2; pyw = y;

//initialize input- and output streams for get and put
WRGET0INIT (ST_INCR, pxr1);  //--initialize input byte-stream0 for x1
WRGET1INIT (ST_INCR, pxr2);  //--initialize input byte-stream1 for x2
WRPUTINIT  (ST_INCR, pyw);   //--initialize output byte-stream for y

for (int i=0; i<VECTOR_LENGTH; i++)
{
   WRGET0I (&A1, NUMBYTE);  //get NUMBYTE from input stream0 to WR A1
   WRGET1I (&A2, NUMBYTE);   //get NUMBYTE from input stream1 to WR A2
   //Invoke extension instruction as C intrinsics using data in WRs: A1 & A2
   isefFoo (A1, A2, &B);
   WRPUTI (B, NUMBYTE);      //put NUMBYTE from WR B to output stream
}
WRPUTFLUSH0 (); //complete transfer with 2 flush instructions
WRPUTFLUSH1 ();
```

2.3.2 Area Performance on Stretch S6 CPU

We investigate the capabilities of reconfigurable fabric on the Stretch S6 ISEF for implementing user-defined custom-precision operations as extension instruction, specifically focusing on custom-precision floating-point arithmetic operations. Our goal is to verify the assumption of superlinear area gain for lower-precision arithmetic operations. Two extension instructions are implemented on the Stretch S6 ISEF: a custom-precision integer multiplication and a custom-precision floating-point fused multiply-accumulate operation.

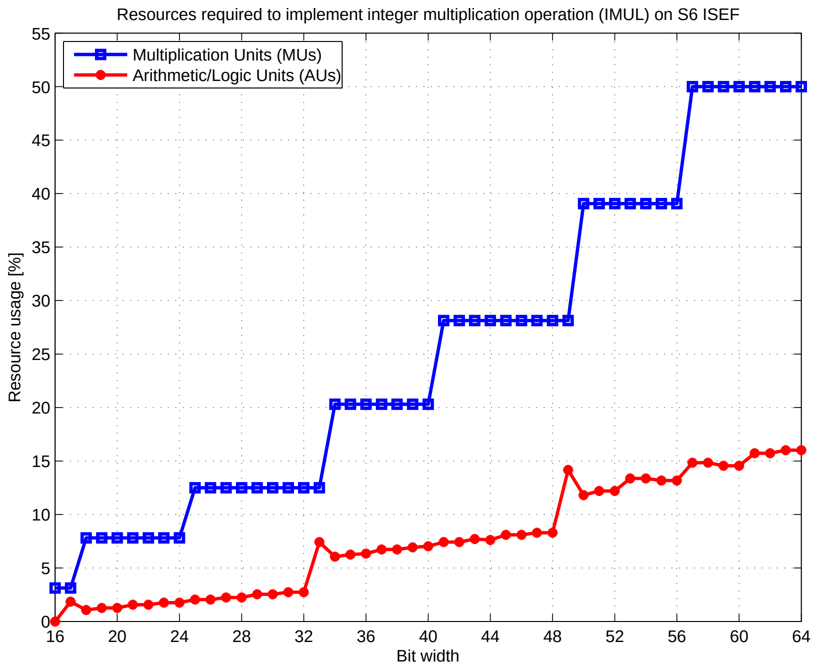


Figure 2.3 S6 ISEF resource usage of IMUL versus precision

Custom-Precision Integer Multiplication Extension Instruction

We start our investigation by implementing a custom-precision integer multiplication (IMUL) of two integer input operands on the ISEF. The precision (bit width) of two input operands is varied in unit step from 16 bits up to 64 bits. We target the ISEF to run at a clock frequency of 150 MHz, half of the clock frequency of the Xtensa core (300 MHz). The Stretch-C code for implementation of a 16-bit IMUL operation is shown below. The extension instruction `IMUL` receives two input operands (`x1` and `x2`) from two WRs and puts the computed result (`y`) back into another WR.

```
//**************************************************
// Custom Precision Integer Multiplication: IMUL16
// PRECISION (bit-width) =   16 (bits)
//**************************************************
#include    <stretch.h>
#define     BITWIDTH    16   // bitwidth = 16 (bits)

SE_FUNC void IMUL16 (WR x1, WR x2, WR *y)
{
   se_uint<BITWIDTH>     A, B;
   se_uint<2*BITWIDTH>   C;

   A = (se_uint<BITWIDTH>) x1 (BITWIDTH-1,0);
   B = (se_uint<BITWIDTH>) x2 (BITWIDTH-1,0);
   C = A*B;

   *y = C;
}
```

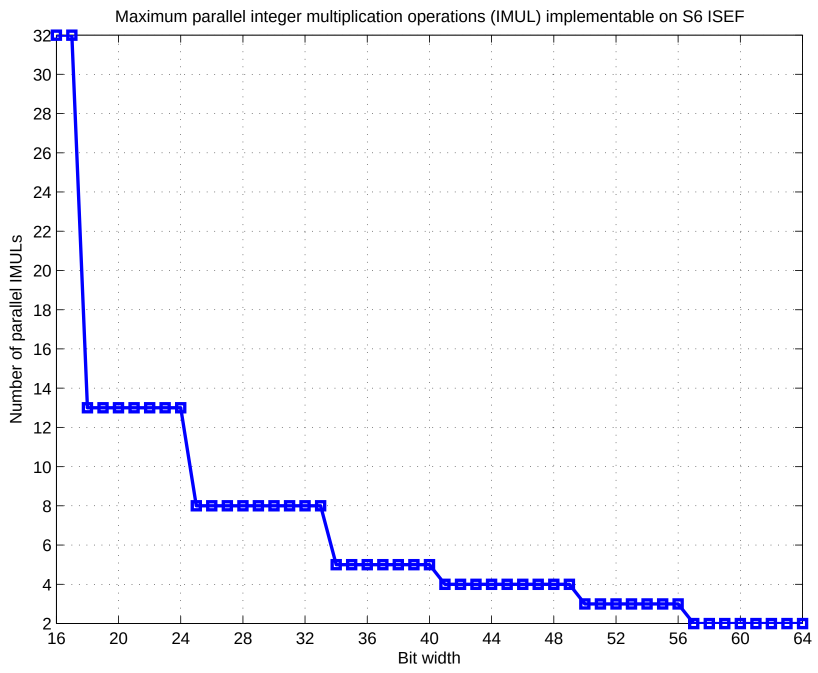


Figure 2.4 Estimated maximum number of parallel IMULs implementable on S6 ISEF versus precision

Figure 2.3 presents the resource usage (in percentage), reported by SCCS, for the IMUL implementation on the Stretch S6 ISEF as a function of precision. The amount of arithmetic/logic units (AUs) required for implementing IMUL increases roughly linearly with the operand precision, while the amount of multiplier units (MUs) required follows a step function.

Using the number of MUs as the dominant resource measure, the maximum number of parallel IMULs that can simultaneously be put into the Stretch S6 reconfigurable fabric is roughly estimated by dividing the total multiplier units available for the amount of resources required by one IMUL, assuming that ISEF is featured with sufficient routing resources. This is demonstrated by Figure 2.4, in which up to 32 parallel 16-bit IMULs can theoretically be put in the ISEF.

Custom-Precision Floating-Point Fused Multiply-Accumulate Extension Instruction

The Xtensa core is equipped with one IEEE single-precision FPU. Our aim is – using the S6's reconfigurable fabric – to provide native floating-point arithmetic of any desired number-format, reusing the remaining logic resources for additional parallel units.

The fused multiply-accumulate (FMA) operation performs $Z = X_1 \cdot X_2 + X_3$ with a single rounding as an indivisible operation, thereby providing a more accurate result compared to multiplication, rounding, addition and rounding. An FMA can therefore potentially increase the accuracy and performance of many computations involving the accumulation of products (e.g. dot product, matrix multiplication or Newton's iteration for function approximation). Figure 2.5 depicts the textbook implementation [49] of an FMA operation as used in this work.

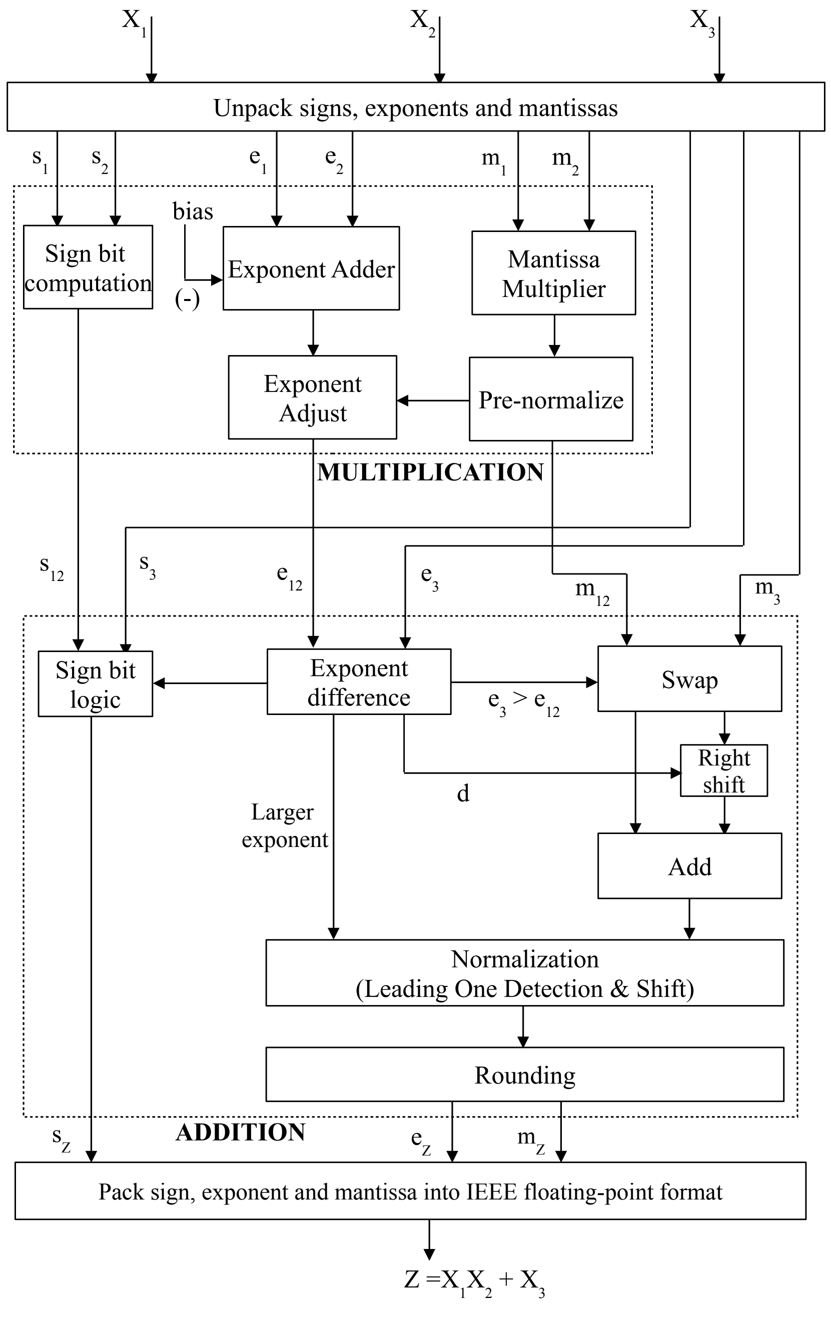


Figure 2.5 Block diagram of floating-point fused multiply-accumulate operation

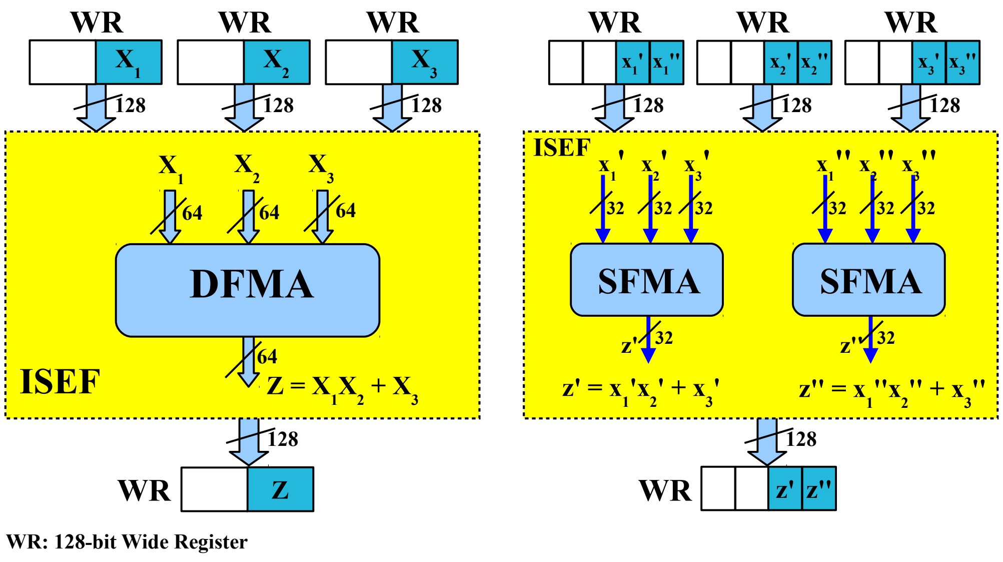


Figure 2.6 a) The `DFMA` and b) multiple parallel `SFMA` extension instructions implemented on S6 ISEF

DFMA and SFMA extension instructions. Using Stretch-C, we specify a double-precision floating-point FMA, namely `DFMA`, as an extension instruction on the ISEF. Figure 2.6a) depicts how the `DFMA` extension instruction is implemented in the ISEF, as well as the way the inputs and output are aligned in the wide registers. The `DFMA` extension instruction accepts three input operands from three 128-bit wide registers, and places the corresponding computed result $Z = X_1 \cdot X_2 + X_3$ in one 128-bit wide register. Since the double precision number format is 64 bits wide, only half of each wide register is used.

To demonstrate the effects of lower-precision operators, we implement a single-precision FMA operator (`SFMA`). The lower resource usage allows for implementation of multiple parallel `SFMA` in the ISEF. Figure 2.6b) depicts how the `SFMAx` extension instruction is implemented in the ISEF, as well as the way the inputs and output are aligned in the wide registers.

Table 2.2 reports the `DFMA` ISEF resource usage as reported by SCCS. It gives the theoretical `DFMA` peak performance of 600 and 80 MFlop/s if executed at 300 and 40 MHz, respectively (assuming a throughput of one `DFMA` or two basic floating point operations per clock cycle). For single-precision number format, Table 2.2 reports the ISEF resource usage when one, two, three or four `SFMA` operators are implemented in parallel. The ISEF routing resources, which are not explicitly reported by the SCC, are exhausted with the implementation of two `SFMAs`, making actual implementation of three or four parallel units impossible.

Choice of clock frequency. The `DFMA` and `SFMA` extension instructions were specified using Stretch C and compiled using SCCS. Obviously, the goal is to achieve an ISEF implementation at the S6's maximum clock frequency of 300 MHz. The compiler, however, could not meet a target frequency of 300 MHz. Given the limited ISEF resources, the FMA's most critical path can be implemented at clock frequencies equal or less than 100 MHz for both the Xtensa core and the ISEF, only. At 100 MHz, SCC was able to synthesize the `DFMA` and `SFMAx` extension instructions (`x` is up to 3).

Table 2.2 Resources required to implement `DFMA`, `SFMA` and `SFMAx` extension instructions on S6 ISEF

	Available	`DFMA`	`SFMA`	`SFMA2`	`SFMA3`	`SFMA4`
Arithmetic/Logic Units (AUs)	4096	54%	21%	42%	62%	84%
Multiplication Units (MUs)	8192	39%	8%	16%	24%	32%
ISEF Stages	27	10	6	7	17	n/a
Routing		routable	routable	routable	not routable	not routable
f_{max}[MHz]	300	100	100	100	100	n/a
Peak performance [MFlop/s] @ f_{ISEF}=300 MHz		600	600	1200	1800	2400
Peak performance [MFlop/s] @ f_{ISEF}=40 MHz		**80**	**80**	**160**	240	320

However, setting the clock frequency of the S6 PCIe (Peripheral Component Interconnect express) board to 100 MHz is currently not possible. The next lower available clock frequency is 40 MHz.

Summary

It is shown that the reconfigurable fabric available on the Stretch S6 CPU can be extended to natively support double-precision floating-point arithmetic operation. For single-precision, even multiple parallel units can be implemented, for which up to four SFMAs can theoretically be deployed on the ISEF. For custom-precision integer multiplication implementation, reducing precision brings a superlinear increase in the number of implementable integer arithmetic units on the ISEF. These two implementations verify well the assumption of superlinear gain in area for lower-precision arithmetic operations on hybrid reconfigurable CPUs.

2.3.3 LINPACK Performance on Stretch S6 CPU

After having shown that double-precision floating-point operations can be implemented on the ISEF, we are concerned with the question how well a complex extension instruction integrates into an existing application. Since the achievable sustained performance of an application running on a hybrid reconfigurable CPU relies heavily on the static interface between reconfigurable fabric and fixed CPU, we would like to understand the effect of different interface limitations on the minimum issue rate of extension instructions implemented on the reconfigurable fabric. In the following, we will characterise the LINPACK benchmark, detail the benchmark's mapping onto the S6 hybrid reconfigurable CPU and report performance measurements for different implementation variants.

LINPACK. LINPACK is a well known benchmark to characterise floating-point performance of computers and widely used both in academia and industry [50]. This benchmark measures how fast a computer solves a dense $N \times N$ system of linear equations $Ax = b$. LINPACK does not require the use of specific number formats but demands the solution to achieve a given accuracy. The implementation used in this work is based on a C code available on netlib[1] relying on BLAS (Basic Linear Algebra

[1] http://www.netlib.org/benchmark/linpackc.new

Subprograms) Level-1 routines. While there exist more efficient LINPACK implementations, our aim in this work is not to maximise LINPACK performance, but to demonstrate relative LINPACK performance among various CPUs and at different precision levels when exploiting short-vector SIMD extension instructions implemented on the Stretch ISEF.

LINPACK uses the BLAS routine `DGEFA` to perform the LU decomposition of the squared matrix A with partial pivoting and `DGESL` to solve the given system of linear equations by forward and back substitution. Most of the execution time of LINPACK is spent in `DGEFA`, of which the largest part is spent in the `DAXPY` routine. `DAXPY` performs $\mathbf{y} = \alpha \cdot \mathbf{x} + \mathbf{y}$, i.e., it multiplies a vector $\mathbf{x}$ with a scalar α and accumulates the result in vector $\mathbf{y}$.

Experiment Setup. The following experiments were set up, reflecting implementation choices available on the Stretch S6 hybrid reconfigurable CPU. For each implementation, the floating-point operators in `DAXPY` were replaced by function calls to the respective extension instruction.

1. LDX {DP LINPACK, software-emulated via Xtensa ALU}: Software-emulated DP arithmetic via Xtensa ALU.
2. LD1 {DP LINPACK, Xtensa+ISEF (`DFMA`)}: `DAXPY` uses extension instruction `DFMA`.
3. LSX {SP LINPACK, Xtensa FPU}: `DAXPY` uses the Xtensa SP FPU (no ISEF used).
4. LS1 {SP LINPACK, Xtensa+ISEF (`1SFMA`)}: `DAXPY` uses extension instruction `SFMA`.
5. LS2 {SP LINPACK, Xtensa+ISEF (`2SFMA`)}: `DAXPY` uses extension instruction `2SFMA`.

Mainstream CPUs. In order to compare performance of the Stretch S6 hybrid reconfigurable CPU to desktop CPUs, we repeat the measurements on two desktop CPUs from AMD (AMD Opteron 2439, 2.8 GHz, 105 W) and Intel (Intel Core i7 970, 3.2 GHz, 130 W). On desktop CPUs, the LINPACK code was compiled using gcc version 4.3.2 under Debian 4.3.2-1.1 with optimization flag `-O3`.

LINPACK Code. For the double-precision experiment, we replace the original sequence of DP floating-point multiplication and addition in `DAXPY` by a single `DFMA` extension instruction. For the single-precision experiments, we use either the S6's SP FPU or the `SFMA` and `2SFMA` extension instruction, respectively. The data transfer is implemented using simple load store (external memory read/write and byte-streaming channels (internal memory to ISEF).

The code was compiled using the Stretch C compiler (SCCS) version 2010.01 (built on 5 Feb 2010). Used SCCS flags were `-stretch-effort10` and `-O3`. Both, Xtensa and ISEF were forced to run at a clock frequency of 40 MHz (i.e. $f_{Xtensa} = f_{ISEF} = 40\ MHz$)

Measurements

For every experiment, we measure the total execution time in cycles. The estimated number of floating-point operations at system size N is $(2/3N^3 + 2N^2)$ [50]. The LINPACK performance in floating-point operations per second (Flop/s) is calculated by dividing the number of estimated floating-point operations by the respective LINPACK execution time.

Table 2.3 reports the performance of DP and SP LINPACK benchmarks achieved on Stretch S6 and on desktop CPUs for systems of size N=500. The performance of DP LINPACK using software-emulated floating-point arithmetic via Xtensa ALU is about 0.5 MFlop/s. By providing native DP

Table 2.3 LINPACK Performance efficiency and Power efficiency at N=500

		Sustained performance [MFlop/s]	Peak performance [MFlop/s]	Performance efficiency	Power efficiency [MFlop/W]
DP emulated via S6 ALU	40 MHz	0.5	80	1%	0.1
DP `DFMA` ISEF	40 MHz	12.6	80	16%	2.0
SP on S6 FPU	40 MHz	7.0	80	9%	1.1
SP 1`SFMA` ISEF	40 MHz	10.1	80	13%	1.6
SP 2`SFMAs` ISEF	40 MHz	13.6	160	8%	2.2
DP AMD Opteron 2439	2800 MHz	1570			14.9
DP Intel Core i7	3200 MHz	1560			12.0

floating-point arithmetic through the `DFMA` extension instruction, the performance achieved by DP LINPACK on S6 is 12.6 MFlop/s. This corresponds to a speed-up of about 25 times compared to DP software-emulated LINPACK. The desktop CPUs operating at much higher clock frequencies significantly outperform the Stretch S6 hybrid CPU in raw performance (MFlop/s). Accepting a lower accuracy by using SP arithmetic, the SP LINPACK implementation using the native SP Xtensa FPU achieves a sustained performance of 7.0 MFlop/s at 40MHz. Using an extension instruction implementing one and two SP FMAs in parallel at 40 MHz, achievable LINPACK performance becomes 10.1 MFlop/s and 13.6 MFlop/s, respectively. This is about 1.5 and 2 times more efficient than SP LINPACK using the native SP Xtensa FPU at the same clock frequency.

Columns three and four in Table 2.3 present the performance efficiency and the power efficiency of all LINPACK implementations on S6 CPU. The best performance efficiency is 16% with `DFMA` for DP Linpack, and 13% with `SFMA` for SP Linpack. The maximum power consumption of the PCIe expansion card holding four Stretch S6 CPUs is 25W. We can currently not measure the actual power consumption and therefore assume a worst-case scenario of 25W/4=6.25W per S6 CPU. Dividing sustained performance by power consumption gives a worst-case estimate of power efficiency at N=500 of 2.0 MFlop/W and 2.2 MFlop/W for DP and SP LINPACK implementations on S6 CPU, respectively.

Performance Measure: Cycles per Extension Instruction (CPEI). In analogy to the cycles per instruction (CPI) measure used in CPU design [51], we will characterise the performance of extension instructions quoting the *cycles per extension instruction* (CPEI). It is equivalent to the *extension instruction issue rate.* The CPEI for some program or code section is calculated as the ratio of the executed Xtensa clock cycles n_{Xtensa} and the executed extension instructions n_{ISEF}

To compare the efficiency of Linpack against the best possible performance achievable on the Stretch S6, we calculate the corresponding CPEI for each implementation. Our Linpack implementation performs $n_{500} = 83833333$ Flops at a system size of N=500. Given the equivalent number of floating-point operations per extension instruction (FPEI), we can calculate the CPEI as $\#cycles \cdot \text{FPEI}/n_{500}$.

Inspecting Table 2.4 shows that the CPEI using `DFMA` is 6.4. Using a single `SFMA` results in a CPEI of 7.9 while using two `SFMAs` gives a CPEI of 11.8. For SP Linpack on Xtensa FPU (no ISEF), we observe that the Xtensa FPU performs a single-precision fused multiply-add instruction within the routine `SAXPY`, with respective assembly code `madd.s`, thereby resulting in a FPEI of 2 and leading to a CPI of 11.5.

Table 2.4 LINPACK execution times and CPEI

		Exec. cycles [$\times 10^6$]	Flops per Ext.Instr.	CPEI/CPI
DP emulated via S6 ALU	40 MHz	6312	2	150.6
DP `DFMA` ISEF	40 MHz	267	2	6.4
SP on S6 FPU	40 MHz	483	2	11.5
SP 1`SFMA` ISEF	40 MHz	332	2	7.9
SP 2`SFMAs` ISEF	**40 MHz**	**247**	4	11.8

Discussion. The S6's reconfigurable fabric is able to provide support for complex floating point operators like fused multiply accumulate (FMA). For double-precision LINPACK, this leads to a speed-up of 25 compared to software-emulated double-precision arithmetic. For single-precision LINPACK, the fused operation outperforms the implementation relying on the Xtensa FPU. A SIMD unit providing two `SFMA` operators in parallel improves performance by about 35%. The desktop CPUs running at clock frequencies of 2.8 and 3.2 GHz outperform the S6 by about two orders of magnitude with respect to throughput. Energy efficiency is about one order of magnitude better on desktop CPUs. The two most evident reasons for the S6's poor performance are the artificially low clock frequency (40 MHz due to a setting issue) and the S6's low extension instruction issue rate.

2.3.4 S6 ISEF Interface Performance Characterisation

We have outlined the FMA implementations and made naive assumptions about achievable peak throughput. LINPACK performance figures reported in Tables 2.3 and 2.4 showed that sustained performance achieved by the benchmark are significantly lower. This section details the interface between ISEF and S6 on-chip memories. We derive theoretical peak throughput from architectural features and present respective measurements. Detailed understanding of the ISEF's interface allows for a better explanation of the observed LINPACK performance as well as a detailed documentation of inherent performance degradation due to S6 architectural limitations.

Bandwidth Requirements. A key feature of reconfigurable fabrics is the fact that some arithmetic operations' complexity increases superlinear with the precision [13, 46, 49]. When reusing freed resources for additional parallel units, reducing precision can therefore lead to superlinear parallelism with decreasing precision. Superlinear parallelism, however, leads to increased total required bandwidth (i.e., if a double-precision unit can be replaced by four single-precision units, each accepting two operands, the total required bandwidth increases from 1*64 bit to 4*32=128 bit). Exploitation of the increased parallelism is subject to availability of this bandwidth. We are, therefore, interested in understanding all effects (both architectural and compiler-induced) that influence the achievable data transfer bandwidth to and from the ISEF using the byte-streaming mechanism.

ISEF Interface. The S6's execution unit connects on-chip memory (D-Cache and DataRAM) and ISEF via 128-bit wide buses. Wide registers (WRs) act as interface for data transfer to and from the ISEF. We are interested in understanding the implications of (i) the number of input WRs used, (ii) the size and number of operands used on achievable throughput of custom SIMD extension instructions using the streaming interface on S6 CPU. In the following, we derive the *minimum* CPEI (CPEI_{min}) for different extension instruction configurations from architectural features and perform experiments to obtain the respective *average* CPEI (CPEI).

Table 2.5 S6 minimum cycles per extension instruction

WRs In	WRs out	Instructions required	$\mathrm{CPEI}_{\mathrm{min}}$
1	0	1 ext. instruction + 1 WR load	1
0	1	1 ext. instruction + 1 WR store	1
1	1	1 ext. instruction + 1 WR load + 1 WR store	2
2	1	1 ext. instruction + 2 WR loads + 1 WR store	3
3	1	1 ext. instruction + 3 WR loads + 1 WR store	4

S6 Minimum CPEI ($\mathrm{CPEI}_{\mathrm{min}}$). The on-chip memory system and the wide register (WR) file are linked with a single 128-bit wide data bus, allowing for a load or a store of at most one 128-bit WR every clock cycle. The Stretch S6 fixed CPU design is an Xtensa LX dual-issue core whose execution unit is able to issue two instructions every clock cycle. As a consequence of the dual-issue architecture, an extension instruction and a WR load or a WR store can be issued simultaneously. Therefore, if an extension instruction consumes (or writes) only a single WR, the absolute minimum issue rate is 1. Every additional WR load or store operation increases the CPEI by one.

The $\mathrm{CPEI}_{\mathrm{min}}$ for all selected S6 extension instruction configurations is reported in Table 2.5. In summary, an S6 extension instruction reading and writing into two different wide registers can be issued every two Xtensa clock cycles, in case all data is accessible in local memory. Every additionally used register increases the $\mathrm{CPEI}_{\mathrm{min}}$ by one clock cycle.

Experiments

We setup a small test program to measure the average $\mathrm{CPEI}_{\mathrm{min}}$ achievable as a function of the number of input wide registers and operand size. *Byte-streaming channels* are used for efficient data transfer to and from the ISEF. The Stretch S6 CPU supports up to three input byte-streams and one output byte-stream (cf. Section 2.3.1). For ease of understanding and presentation, we implement two simple extension instructions `DNEGx` and `SNEGx`, performing negation for double-precision and single-precision floating-point operands, respectively. For each extension instruction, there exist variants reading data from the lower part of one (`DNEG1, SNEG1`), two (`DNEG2, SNEG2`) or three (`DNEG3, SNEG3`) wide registers. The result of the operation is always a single floating-point number. The extension instruction is executed in a loop reading data from on-chip or off-chip memory. Note, that in C/C++ programs we declare WRs as local variables within the main function for better performance. For our design of negation operations on ISEF, the Xtensa core runs at 300 MHz, and the Xtensa core and ISEF were forced to run at the same clock frequency by using SCCS flag `-stretch-issue-rate 1`, i.e. $f_{\mathrm{ISEF}} = f_{\mathrm{Xtensa}} = 300$ MHz.

On-Chip Memory Access

For on-chip memory access, byte-streaming channels are used. For each extension instruction (i.e. using 1, 2 or 3 input WRs), the input operand size is varied between 32, 64, 96 and 128 bit. The resulting sustained CPEIs of the SIMD extension instructions using byte-streaming channels when the data reside in on-chip memories are reported in Table 2.6. The measured CPEIs when data is in D-Cache and in DataRAM are almost equal. Therefore only the smaller measured CPEI is chosen and reported.

For data transfer between on-chip memory and wide registers via byte-streaming channels, the

Table 2.6 Average CPEI for on-chip memory access via byte-streaming channels.

	CPEI_{min}	32 bits ops.	64 bits ops.	96 bits ops.	128 bits ops.
1 input WR, 1 output WR	2	3.04	3.09	3.13	3.18
2 input WRs, 1 output WR	3	3.07	3.17	3.22	3.29
3 input WRs, 1 output WR	4	4.05	4.09	4.14	4.18

CPEI_{min} is expected to depend on the number of WRs used, but not on the size of the operands within a WR. This is confirmed by our measurements reported in Table 2.6. For configurations using two and three input WRs, the measured CPEI almost matches the CPEI_{min}. For configuration using one input WR, the measured CPEI is one cycle more than the expected CPEI_{min}, which is due to a compiler limitation on unrolling the loop of extension instructions.

Discussion

Hybrid reconfigurable CPUs are prime candidates for power-efficient acceleration of demanding signal/speech processing applications. Reconfigurable fabrics can provide superlinear parallelism when implementing short-vector SIMD units for selected arithmetic operations in reduced precision. This genuine advantage of reconfigurable logic can only be exploited in reconfigurable hybrid CPUs if the interface between reconfigurable logic and fixed CPU can provide the necessary bandwidth for data transfer. In this chapter, we have explored the data bandwidth of the interface between reconfigurable fabric and fixed CPU of the Stretch S6 hybrid reconfigurable CPU. We derived minimum cycles per extension instruction between 1 and 4, depending on the number of WRs used. The streaming channels work as expected, decoupling extension instruction issue rate from the amount of bits consumed by each WR. Our CPEI measurements confirm the expected minimum cycle values.

The Stretch S6 features a large and versatile reconfigurable fabric with impressive I/O (3x128 bit in, 2x128 bit out). The surrounding infrastructure does not match these capabilities, however, limiting the overall data transfer to 128 bit per clock cycle. Fast floating-point arithmetic relies on efficient transfer of large operands. The multiple units implementable in the ISEF cannot be fed with the necessary data, resulting in frequent stalls and inefficient program execution. Benchmarking LINPACK using a floating point FMA extension instruction showed the functional viability of using the S6 for scientific workloads, but achieved disappointing performance figures. These low figures were due to a low clock frequency of 40 MHz (compared to achievable 100 MHz for the extension instruction and a maximum clock rate of 300 MHz for the Xtensa core), the limited I/O bandwidth between ISEF and on-chip memory and the off-chip memory latency.

2.4 Conclusions

This chapter has investigated the area performance and throughput performance of floating-point arithmetic operations on different hardware platforms including conventional CPUs, GPUs, FPGAs and hybrid reconfigurable CPUs. It is known that reduced precision in floating-point arithmetic operations directly translates into gains in peak performance (by a factor of 2 to 12 depending on specific hardware architecture). Besides, on reconfigurable hardware (i.e., FPGA-based platforms) reduced precision allows for increased parallelism.

More specifically, we focused on a prototypical hybrid reconfigurable CPU - the Stretch S6 - as a case study. We have shown that the reconfigurable fabric of a Stretch S6 CPU is able to provide native support for custom-precision floating-point arithmetic up to an IEEE-754 (partly) compatible double-precision number format. For single-precision, multiple operators can be implemented in parallel, theoretically up to four `SFMAs` can be put in the S6 ISEF. This generally allows for trading accuracy with parallelism and performance. The dominant issue identified in this chapter while investigating the Stretch S6 CPU is the mismatch between reconfigurable fabrics and I/O bandwidth available on hybrid reconfigurable CPUs, thereby resulting in low LINPACK performance figures and making hybrid reconfigurable CPUs unsuitable for scientific workloads.

The *precision-to-performance* relation presented in this chapter is the motivation for custom-precision floating-point implementations of signal processing algorithms on reconfigurable hardware in general. Besides, one necessary link to bridge the route from *required accuracy to achieved performance*, the *accuracy-to-precision* relation, is still missing. Having this link established is essential to obtaining an efficient implementation of floating-point algorithms on reconfigurable hardware and will be studied in the remaining chapters of the thesis.

3 Floating-Point Error Analysis Using Affine Arithmetic: Theory

3.1 Introduction

Error analysis is concerned with understanding and estimating the effect of parameter variations and rounding error on an algorithm's final result. In this work, we focus on the effects of rounding error in floating-point arithmetic computations. Numerical rounding error analysis deals with the question how the operations of some given algorithm implemented in finite-precision floating-point arithmetic affect the final numerical result.

There exist different ways to classify error analysis techniques. One way is dividing those techniques into static approaches and dynamic approaches [24]. Static error analysis is based on analytical derivations, often providing conservative estimates. Dynamic error analysis is based on simulations and gives typically more realistic error estimates but requires typically long simulation times and the validity depends on the chosen test data.

For floating-point arithmetic, methods for static floating-point error analysis comprise the conventional error analysis and the range-based error analysis techniques. The conventional forward error analysis often provides very pessimistic error estimates rendering this method impractical for bit width allocation. A comprehensive discussion on the conventional error analysis for floating-point algorithms is presented in [5]. The range-based error analysis techniques often make use of Interval Arithmetic (IA) and Affine Arithmetic (AA). Examples for range-based error analysis techniques are given in [3, 4, 24, 25, 28].

This chapter aims at presenting the fundamentals of using AA for floating-point error modeling and estimation. They are the basis for the implementation of a Matlab-based tool for AA-based automatic floating-point error analysis presented in Chapters 4 and AA-based error analysis of some floating-point signal processing algorithms presented in Chapter 5.

The remainder of this chapter is structured as following. We briefly present some background on floating-point arithmetic, specifically focused on the IEEE-754 standard, and give an introduction of IA and AA. We then present the AA-based error models for floating-point number and unary/binary operations. Using AA we suggest an AA-based error model for a floating-point fused multiply-accumulate (FMA), which is one of the thesis contributions. The chapter closes with some concluding remarks.

3.2 Background

3.2.1 Floating-Point Arithmetic

Floating-point arithmetic is the standard approach for approximating real number arithmetic in modern computers. We denote $\mathbb{R}$ as the set of real numbers and $\mathbb{F}$ as the finite set of floating-point numbers. Every real number $x \in \mathbb{R}$ can be approximated by a floating-point representation $\hat{x} \in \mathbb{F}$,

$$x \to \hat{x} : \ \hat{x} = fl(x),$$

where the transformation $fl(\cdot)$ from $\mathbb{R}$ to $\mathbb{F}$ is called *rounding*. A floating-point number uses three fields to represent a real number: a sign bit s, a biased exponent e and a mantissa f. Given s, e, f, the bias[1], and precision p, using IEEE-754 conventions, a normalised binary floating-point numer evaluates to $(-1)^s \cdot 2^{e-\text{bias}} \cdot 1.f$; where the mantissa $1.f$ is represented with p bits.

Due to the limited amount of bits used for storing a floating-point number in computers, there exists a representation error

$$x - fl(x),$$

between the real number x and its approximation in the floating-point number format. When evaluating a single floating-point arithmetic operator, its respective floating-point operation is affected by an evaluation error. For a sequence of floating-point operations, the representation and evaluation errors are propagated over that sequence. In this work, we refer to these two types of errors - representation error and evaluation error - as *rounding errors*.

Given the working precision p, an important question is to quantify the rounding error of floating-point arithmetic in a convenient form. Higham [5] presents a simple model for estimating the rounding error of floating-point arithmetic, which is refered as *conventional error model* in this thesis.

Conventional error model of floating-point numbers [5]. The conventional error model describes the relation between a real number x and its approximated floating-point representation $\hat{x}$ in which the relative error δ of the approximation $\hat{x} = fl(x)$ is no larger than the unit roundoff $u = 2^{-p}$ as

$$\hat{x} \quad = \quad x(1+\delta), \quad |\delta| \leq u. \tag{3.1}$$

The absolute rounding error of a floating-point number is defined by

$$x - fl(x) = x - \hat{x} = x\delta, \tag{3.2}$$

and obviously the absolute rounding error depends on both the working precision and the magnitude of the number itself. Compared to fixed-point numbers, floating-point numbers have a fixed relative error and cover a larger dynamic range using less bits. The advantages of floating-point arithmetic come at the cost of non-uniform resolution and more complex hardware implementation [49].

In this work, we prefer using the following expression

$$\hat{x} \quad = \quad x + x \cdot \delta, \quad |\delta| \leq u, \tag{3.3}$$

which is equivalent to (3.1), for representing the conventional error model of floating-point number. The implication of this alternative representation for the conventional floating-point error model in (3.3) will be explained later in Section 3.3.

[1]By using a bias, the exponent field in a floating-point number is an unsigned number making it easier for the implementation of floating-point operations, e.g., performing the comparison of floating-point numbers [49].

Conventional error model of floating-point arithmetic operations [5]. The rounding error due to finite-precision computation of a floating-point arithmetic operation is expressed by the conventional error model as follows

$$fl(x \text{ op } y) \quad = \quad (x \text{ op } y)(1+\delta), \quad |\delta| \le u, \quad \text{op} = +, -, \times, /. \tag{3.4}$$

It is common to assume that (3.4) holds also for the square root operation.

Propagation of rounding errors. Rounding errors are propagated during a sequence of floating-point arithmetic operations. Conventional error propagation (as used by [5]) tries to derive symbolic expressions in δ using the models in (3.1) and (3.4) and typically concentrating on higher order terms, often leading to a final error expression consisting of high order terms in δ. As a consequence, the rounding error bound derived by the conventional error model is very conservative and often depends on one or several orders of the problem size, making conventional error analysis unsuitable for bit width allocation. thus making conventional error analysis unsuitable for bit width allocation.

3.2.2 Interval Arithmetic

Interval Arithmetic, also known as *interval analysis*, was invented in the 1960's by Ramon E. Moore [52] for solving range problems. IA is a range-based model for numerical computations where each real quantity x is represented by an interval $\bar{x} = [a, b]$. Those intervals are added, subtracted, multiplied, etc., in such a way that each computed interval $\bar{x}$ is guaranteed to cover the unknown value of the corresponding real quantity x. The addition, subtraction, multiplication and division operations for two intervals $\bar{x} = [a, b]$ and $\bar{y} = [c, d]$ are performed by equations (3.5 - 3.8). Note that these expressions ignore roundoff errors. Refer to [53] for extensive information on IA.

$$\bar{x} + \bar{y} \quad = \quad [a + b, \ c + d] \tag{3.5}$$

$$\bar{x} - \bar{y} \quad = \quad [a - d, \ b - c] \tag{3.6}$$

$$\bar{x}\bar{y} \quad = \quad [\min(ac, ad, bc, bd), \ \max(ac, ad, bc, bd)] \tag{3.7}$$

$$\bar{x}/\bar{y} \quad = \quad [\min(a/c, a/d, b/c, b/d), \ \max(a/c, a/d, b/c, b/d)] \tag{3.8}$$

In error analysis, IA can be used to derive worst-case rounding error bounds. One example of IA for error anlysis is the work of Krämer [3] deriving a-priori worst case error bounds for basic floating-point operations. The major disadvantages of IA are twofold. First, IA cannot capture the correlations between variables, thereby possibly leading to range overestimation. As a typical example, one could use IA to estimate the range of the expression $x - x$, with original IA range $\bar{x} = [-1, \ 1]$. The estimated result is then calculated as $\bar{x} - \bar{x} = [-2, \ 2]$, which is twice as wide as the original range $\bar{x}$, and quite far from the expected range of $[0, \ 0]$, which is the true value. Second, the range evaluation is done immediately at every computation step, thus sequential information of a long computational chain cannot be kept. As a result, overestimation may be accumulated along the computational chain, leading to range explosion.

3.2.3 Affine Arithmetic

Affine Arithmetic (AA), also known as *affine analysis*, was introduced in [27] by Comba and Stolfi in 1993. It was then presented in more detail in [53], including the pseudocode for the implementation of

AA. AA is an improvement of IA in the sense that it can keep track of correlations between quantities, thereby potentially leading to more accurate estimated ranges compared to the ones computed by IA. In AA, the uncertainty of any real quantity x is represented by an affine form $\hat{x} = g(\epsilon_1, ..., \epsilon_n)$, which is a first-order polynomial expressed as

$$\hat{x} = x_0 + x_1\epsilon_1 + ... + x_n\epsilon_n = x_0 + \sum_{i=1}^{n} x_i\epsilon_i, \tag{3.9}$$

where the coefficient x_0 is the central value, while the other coefficients x_i are partial deviations ($i = 1, 2, ..n$). The ϵ_i are noise symbols, whose values are unknown but assumed to lie in the interval $U = [-1, 1]$. Each noise symbol ϵ_i stands for an independent uncertainty component of the total uncertainty of the quantity x, and the corresponding coefficient x_i gives the magnitude of that component.

There are two types of arithmetic operations in AA: affine operations and non-affine operations. Affine operations include add, subtract and multiply with a constant; non-affine operations are the remaining ones. Affine operations can accurately be modeled, while non-affine operations have to be approximated. Given two AA forms

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i\epsilon_i, \quad \hat{y} = y_0 + \sum_{i=1}^{n} y_i\epsilon_i,$$

some basic operations on $\hat{x}$ and $\hat{y}$ are performed as follows

$$\begin{aligned}
\hat{x} \pm \hat{y} &= x_0 \pm y_0 + \sum_{i=1}^{n}(x_i \pm y_i)\epsilon_i, && (3.10)\\
C\hat{x} &= Cx_0 + \sum_{i=1}^{N} Cx_i\epsilon_i, && (3.11)\\
\hat{x}\hat{y} &= x_0y_0 + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\epsilon_i + \; B(\sum_{i=1}^{n} x_i\epsilon_i) \cdot B(\sum_{i=1}^{n} y_i\epsilon_i) \cdot \epsilon_{n+1}, && (3.12)
\end{aligned}$$

where C in (3.11) is a constant. Refer to [53] for a complete reference of AA.

The key feature of AA is that the same noise symbol may contribute to the uncertainty of two or more quantities, meaning that the AA model is able to keep track of correlations between quantities, offering the possibility of component cancelation, and, therefore, the possibility of more accurately estimated ranges. The drawback of AA, however, is that the linearity will not be guaranteed for non-affine operations often requring non-linear approximates, like Chebyshev approximation [53], thereby leading to overestimated ranges for non-affine operations. Another drawback of AA is that AA analysis is more expensive than IA since the entire expressions (often proportional in length to the number of operations) are maintained rather than just intervals.

Affine Arithmetic vs. Interval Arithmetic. We perform comparisons between AA and IA through two simple examples. Figure 3.1 describes the first example for range estimation of $z = 2x + 3y$ using AA and IA. We assume that the two input quantities are given in the ranges $x \in [6, 14]$ and $y \in [18, 22]$, and, in their AA representations, share the same noise symbol ϵ_1 as

$$\begin{aligned}
\hat{x} &= 10 + 3\epsilon_1 + 1\epsilon_2,\\
\hat{y} &= 20 - 2\epsilon_1.
\end{aligned}$$

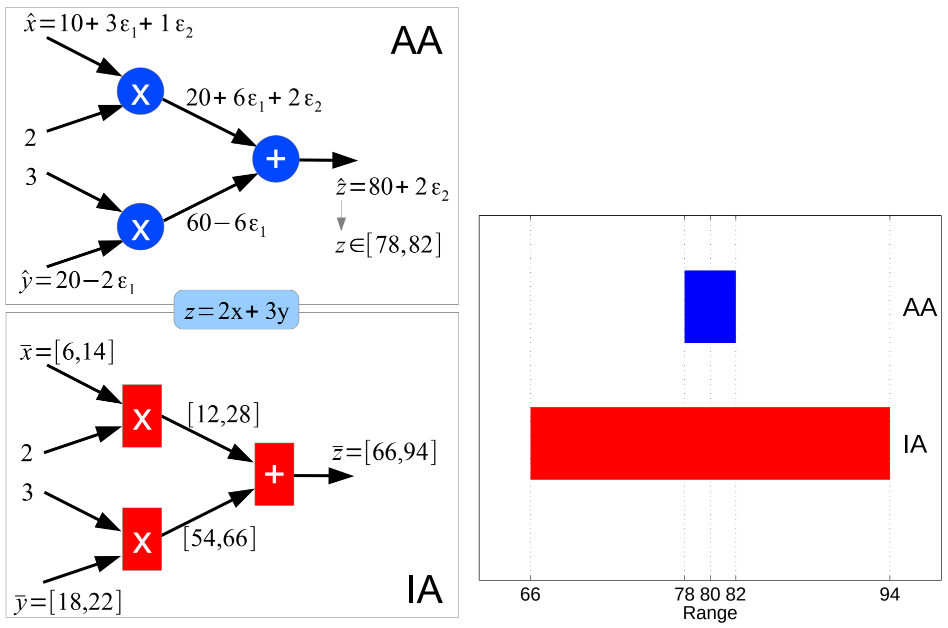


Figure 3.1 The first example of range estimation using AA and IA

The AA evaluation of $\hat{z} = 2\hat{x} + 3\hat{y}$ gives the final AA form $\hat{z} = 80 + 2\epsilon_2$, for which the noise symbol ϵ_1 is cancelled, resulting in the corresponding range $[78, 82]$ for z. The range estimated by AA is, therefore, seven times smaller in comparison with the range estimated by IA, i.e., $\bar{z} = [66, 94]$, with which the correlation between x and y is not considered (see Figure 3.1). In fact, this example consists only affine operations, i.e., multiplication with constant and addition, therefore the modeling errors due to non-linear approximation do not happen.

In the second example, we compute one output z in two different ways from the inputs x and y using IA and AA, leading to a total of four different intervals for the output given the intervals of the inputs. Given $x \in [2, 5]$ and $y \in [7, 9]$, we would like to compute

$$\begin{aligned} z_1 &= (x+y)^2 - (x-y)^2 \\ z_2 &= 4(xy). \end{aligned}$$

For evaluating the intervals of z_1 and z_2, we use equations (3.5 - 3.7) for IA and equations (3.10 - 3.12) for AA. First, we decompose z_1 and z_2 in elementary operations

- for z_1:

$$\begin{aligned} u_1 &= x + y, \\ u_2 &= x - y, \\ u_3 &= u_1 \cdot u_1, \\ u_4 &= u_2 \cdot u_2, \\ z_1 &= u_3 - u_4, \end{aligned}$$

❏ for z_2:

$$\begin{aligned} v_1 &= x \cdot y, \\ z_2 &= 4 \cdot v_1. \end{aligned}$$

Using IA for computing z_1 and z_2 gives us

❏ for z_1:

$$\begin{aligned} \bar{u}_1 &= [\min(\bar{x}) + \min(\bar{y}),\ \max(\bar{x}) + \max(\bar{y})] = [9, 14], \\ \bar{u}_2 &= [\min(\bar{x}) - \max(\bar{y}),\ \max(\bar{x}) - \min(\bar{y})] = [-7, -2], \\ \bar{u}_3 &= [\min^2(|\bar{u}_1|),\ \max^2(|\bar{u}_1|)] = [81, 196], \\ \bar{u}_4 &= [\min^2(|\bar{u}_2|),\ \max^2(|\bar{u}_2|)] = [4, 49], \\ \bar{z}_1 &= [\min(\bar{u}_3) - \max(\bar{u}_4),\ \max(\bar{u}_3) - \min(\bar{u}_4)] = [\mathbf{32}, \mathbf{192}], \end{aligned}$$

❏ for z_2:

$$\begin{aligned} \bar{v}_1 &= [\min(\bar{x}) \cdot \min(\bar{y}),\ \max(\bar{x}) \cdot \max(\bar{y})] = [14, 45],\ (x, y > 0) \\ \bar{z}_2 &= [4\min(\bar{v}_1),\ 4\max(\bar{v}_1)] = [\mathbf{56}, \mathbf{180}]. \end{aligned}$$

Using AA, we assume the AA forms for $x \in [2, 5]$ and $y \in [7, 9]$ as

$$\begin{aligned} \hat{x} &= 3.5 + 1.5\epsilon_1 \qquad \text{with } \epsilon_1 \in [-1, +1], \\ \hat{y} &= 8 + 1\epsilon_2 \qquad\quad \text{with } \epsilon_2 \in [-1, +1], \end{aligned}$$

and compute the ranges for z_1 and z_2 as follows:

❏ for z_1:

$$\begin{aligned} \hat{u}_1 &= \hat{x} + \hat{y} = 11.5 + 1.5\epsilon_1 + 1\epsilon_2 \quad \in [9, 14] \\ \hat{u}_2 &= \hat{x} - \hat{y} = -4.5 + 1.5\epsilon_1 - 1\epsilon_2 \quad \in [-7, -2] \\ \hat{u}_3 &= \hat{u}_1\hat{u}_1 \quad = 132.25 + 34.5\epsilon_1 + 23\epsilon_2 + 6.25\epsilon_3 \quad \in [68.5, 196] \\ \hat{u}_4 &= \hat{u}_2\hat{u}_2 \quad = 20.25 - 13.5\epsilon_1 + 9\epsilon_2 + 6.25\epsilon_4 \quad \in [-8.5, 49] \\ \hat{z}_1 &= \hat{u}_3 - \hat{u}_4 = 112 + 48\epsilon_1 + 14\epsilon_2 + 6.25\epsilon_3 - 6.25\epsilon_4 \quad \in [\mathbf{37.5}, \mathbf{186.5}] \end{aligned}$$

❏ for z_2:

$$\begin{aligned} \hat{v}_1 &= \hat{x}\hat{y} = 28 + 12\epsilon_1 + 3.5\epsilon_2 + 1.5\epsilon_5 \quad \in [11, 45] \\ \hat{z}_2 &= 4\hat{v}_1 = 112 + 48\epsilon_1 + 14\epsilon_2 + 6\epsilon_5 \quad \in [\mathbf{44}, \mathbf{180}] \end{aligned}$$

A summary of the four different intervals computed for the same output z given in Table 3.1. Figure 3.2 presents a comparison among the four different ranges for the output z. We actually know that the true range of z is [56, 180], i.e., the result of IA applied to the algebraically simplified expression $z_2 = 4xy$. Compared to this result, AA has advantages and disadvantages:

Table 3.1 Evaluation of z in two different ways using IA and AA

	z_1	z_2
IA	[32, 192]	**[56, 180]**
AA	[37.5, 186.5]	[44, 180]

$z = z_1 = z_2;\ z_1 = (x+y)^2 - (x-y)^2,\ z_2 = 4(xy)$,
with $x \in [2,5],\ y \in [7,9]$, the true range is [56,180].

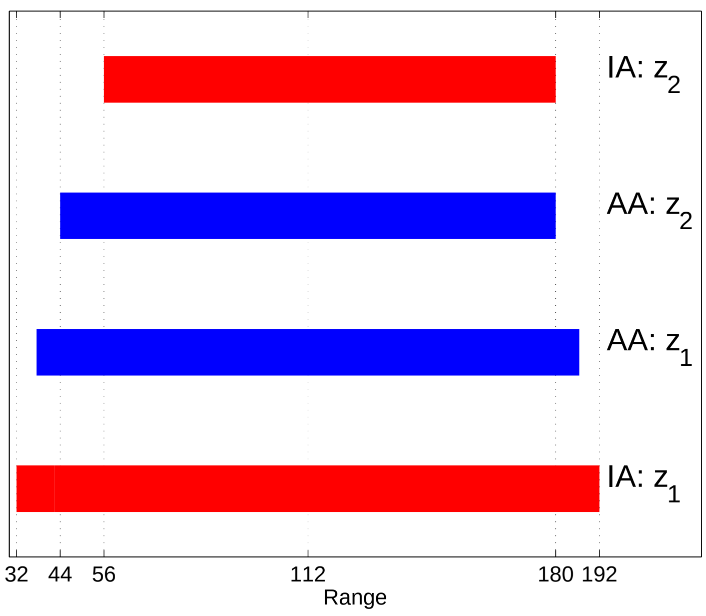


Figure 3.2 The second example of range estimation using AA and IA

- For non-affine (or non-linear) operations such as the multiplication $v_1 = xy$, AA introduces approximation errors due to the restriction of the affine model. These approximation errors are due to the introduction of additional noise symbols which are, however, not independent of the existing ones. For the case of v_1, the approximation error corresponds to the evaluation of the last term in expression (3.12) and it is associated with a new noise symbol ϵ_5.

- For longer cascades of operations as in the computation of z_1, AA memorizes parts of the computational history, which may result in the cancellation of "error" terms, as seen when computing the difference $u_3 - u_4$, where the terms in ϵ_1 and ϵ_2 show up in both u_3 and u_4. This may improve in overall accuracy as illustrated by the observation, i.e., in Table 3.1 and Figure 3.2, that the AA result $z_1 \in [37.5,\ 186.5]$ is closer to the true range $z_2 \in [56,\ 180]$ than the IA result $z_1 \in [32,\ 192]$.

General form of an AA expression. The simple form of an AA expression representing the uncertainty of a quantity x is represented in (3.9). The AA expression $\hat{Z}$ for representing a quantity z that is an output of some numerical operations can generally be written as [4]

$$\hat{Z} = Z_0 + Z_1 \cdot \epsilon_1 + \cdots + Z_N \cdot \epsilon_N = Z_0 + \sum_{i=1}^{N} Z_i \cdot \epsilon_i\ , \quad \epsilon_i \in [-1, 1] \tag{3.13}$$

where Z_0 is the central value and Z_i $(i = 1..N)$ are the constants evaluated by applying the bounding operator. The bounding operators for AA expressions are described below.

Bounding operators. In range-based analyses (IA and AA) we often need to estimate the numerical range of uncertain quantities. This estimation is performed by using *bounding operators* during the computational chain of a numerical algorithm and/or at the final numerical operation for evaluating the range of the final computed result. For non-affine operations in AA, the bounding operator is needed to approximate AA expressions with higher-order noise terms by AA expressions with first-order noise terms. If we denote $B(\hat{Z})$ is the bounding operator, then the lower and upper bounds for $\hat{Z}$ are $(Z_0 - B(\hat{Z}))$ and $(Z_0 + B(\hat{Z}))$, respectively. Two bounding operators can be used: the *hard bounding operator* and the *probabilistic bounding operator* [28]. The hard bounding operator is typically used for IA to estimate the worst-case interval. For AA, both hard and probabilistic bounding operators can be applied.

AA-based hard bounding operator. The upper hard bounding operator is defined for the extreme cases of the noise symbols $\epsilon_i = \pm 1$ and given by

$$B_{hard}(\hat{Z}) = |Z_1| + ... + |Z_N| = \sum_{i=1}^{N} |Z_i|. \tag{3.14}$$

AA-based probabilistic bounding operator. The main drawback of using a hard bounding operator for evaluating an AA expression is the significant decrease in accuracy of estimated bounds compared to the real ranges in practical applications. Fang [4] showed that when the number of noise symbols increases up to $N = 10$, the ratio of the maximum simulated range[2] over the theoretical bound

[2]The maximum range is obtained by conducting Monte Carlo simulations for 10^6 runs.

estimated by using a hard bounding operator is about 75%. This ratio decreases significantly down to 25% and 7.5% when N increases up to 100 and 1000 components, respectively. The reason for this, when using an AA-based hard bounding operator, is that the probability for all noise symbols being equal to the maximal $\epsilon = \pm 1$ is often extremely low.

To overcome this drawback, distribution information of the noise symbols can be incorporated to form a probabilistic bounding operator, as suggested in [28]. We assume that the noise symbols ϵ_i are independently and uniformly distributed random variables (RVs) in $[-1, 1]$, with zero mean and variance $\sigma_i^2 = 1/3$. By setting

$$S_N = \sum_{i=1}^{N} Z_i \cdot \epsilon_i$$

the general AA form for a floating-point computed result in (3.13) can be rewritten as

$$\hat{Z} = Z_0 + \sum_{i=1}^{N} Z_i \cdot \epsilon_i = Z_0 + S_N,$$

where S_N can be seen as a linear combination of N independently and identically distributed (IID) random variables ϵ_i.

According to the *Central Limit Theorem* (CLT), the distribution of S_N will converge to a normal distribution (Gaussian) when N goes to infinity [54]. Note that the CLT makes the statement for a sum of IID random variables. Here the case is somewhat different in the sense that we have a linear combination of IID random variables in S_N. However, if the individual components in the sum are of similar magnitude then the CLT will hold. In fact, experiments show that the distribution of S_N still becomes Gausiian. In practical cases, for a finite number of observations, a reasonable approximation of the normal distribution can be obtained with a sufficiently large N. The standard deviation σ_N (variance σ_N^2) of the distribution of S_N is computed by

$$\sigma_N = \sqrt{\sum_{i=1}^{N} Z_i^2 \sigma_i^2} = \sqrt{\frac{1}{3} \sum_{i=1}^{N} Z_i^2}. \tag{3.15}$$

We then define the probabilistic bound of $\hat{Z}$ as

$$B_{prob}(\hat{Z}) = K \cdot \sigma_N, \tag{3.16}$$

which is a bound with a degree of confidence $K > 0$ ($K \in R^+$) or a bound with confidence interval $K\sigma_N$. As about 99.7% of the values drawn from a normal distribution are within three standard deviations, a practical choice is $K = 3$ for estimating the AA-based probabilistic bound in (3.16) corresponding to a confidence interval of $3\sigma_N$.

Numerical experiments in [4] showed that the distribution of a simple AA form $\hat{x}$ can be approximated by a Gaussian for $N \geq 4$, in that case an AA-based probabilistic bounding operator can be applied. If the number of noise symbols is small, an AA-based hard bounding operator is used instead.

3.3 Floating-Point Error Modeling with Affine Arithmetic

AA can be used to model the rounding errors of floating-point arithmetic computations, as presented in the work of Fang [4]. Our work also applies AA for performing error analysis of floating-point signal processing algorithms. In contrast to Fang's work that considered only linear transforms, this work goes beyond to include nonlinear algorithms, in particular the Levinson-Durbin algorithm presented later in Chapter 5. We aim to estimate the rounding error bound and use that bound for bit width allocation. In this section, we briefly describe how AA is used to model floating-point numbers and floating-point arithmetic operations.

3.3.1 AA-Based Error Model for Floating-Point Numbers

As the uncertainty term δ of the conventional error model for a scalar floating-point number in (3.3) is assumed to lie in the range $[-u, u]$, it can be represented as $\delta = u \cdot \epsilon$ where ϵ is the noise symbol, $\epsilon \in [-1, 1]$; $u = 2^{-p}$ is the unit roundoff, and p is the precision. The equivalent AA model for a floating-point number is rewritten as follow

$$x_f \quad = \quad x + x \cdot u \cdot \epsilon, \quad \epsilon \in [-1, 1], \tag{3.17}$$

where ϵ is the noise symbol representing the uncertainty of the absolute rounding error of x_f due to the finite-precision number format.

AA-based model for floating-point variables. In practice, a physical quantity or signal is often given in some range or it can be represented by a variable. Assuming that a real variable $x \in \mathbb{R}$ is in the range $[x_0 - x_1,\ x_0 + x_1]$ with the central value x_0 and the deviation x_1, the corresponding AA form of the real variable x is

$$\hat{x} = x_0 + x_1\epsilon_1, \quad \epsilon_1 \in [-1, 1],$$

where ϵ_1 is a noise symbol representing the uncertainty in the range of the real variable x.

We are now ready to represent the real variable x in floating-point number format using AA. Simlilar to (3.17), the corresponding floating-point variable $x_f \in \mathbb{F}, x_f = fl(x)$, is modeled by an AA expression $\hat{x}_f$ as

$$\hat{x}_f = \hat{x} + \hat{x} \cdot u \cdot \epsilon_2, \quad \epsilon_2 \in [-1, 1],$$

with a new noise symbol ϵ_2 for representing the uncertainty of the absolute rounding error of floating-point variable x_f. Note that $\hat{x} = x_0 + x_1\epsilon_1$, the AA model for a floating-point variable is then rewritten as follow

$$\hat{x}_f = (x_0 + x_1\epsilon_1) + (x_0 + x_1\epsilon_1) \cdot u \cdot \epsilon_2. \tag{3.18}$$

Range component and Rounding error component. The AA form in (3.18) describes the uncertainty in a floating-point variable x_f as a sum of two components: *i)* the numerical range component $(x_0 + x_1\epsilon_1)$ depending on the noise symbol ϵ_1, and *ii)* the rounding error component $(x_0 + x_1\epsilon_1) \cdot u \cdot \epsilon_2$ depending on both noise symbols ϵ_1 and ϵ_2. Apparently, the rounding error component depends on both the range component and the unit roundoff, as it is a property of any floating-point number.

The rounding error component is a second-order term with respect to the noise symbols, which however violates the requirement of having only first-order noise symbols in an AA representation. To make expression (3.18) comply with the AA representation, a bounding operator $B(\cdot)$ is applied on the range component of the rounding error component, i.e., $B(x_0 + x_1\epsilon_1)$, such that by using a bounding operator the rounding error component can be approximated as a first order term of the noise symbol ϵ_2 only, i.e.,

$$(x_0 + x_1\epsilon_1) \cdot u \cdot \epsilon_2 \approx B(x_0 + x_1\epsilon_1) \cdot u \cdot \epsilon_2,$$

or

$$\hat{x}_f \approx (x_0 + x_1\epsilon_1) + B(x_0 + x_1\epsilon_1) \cdot u \cdot \epsilon_2. \tag{3.19}$$

We denote $\hat{x}_f^r$ and $\hat{x}_f^e$ as the range- and rounding error components, respectively. Using (3.19), the AA model for a floating-point variable x_f is rewritten in a convenient representation as

$$\hat{x}_f \quad = \quad \hat{x}_f^r + \hat{x}_f^e, \tag{3.20}$$

with

$$\begin{aligned} \hat{x}_f^r &= (x_0 + x_1\epsilon_1), & (3.21)\\ \hat{x}_f^e &= B(\hat{x}_f^r) \cdot u \cdot \epsilon_2. & (3.22) \end{aligned}$$

The bounding operator $B(\cdot)$ can be a hard bounding operator B_{hard} defined in (3.14) or a probabilistic bounding operator B_{prob} defined in (3.16). The convenient model obtained with a bounding operator in these expressions above, however, comes at the cost of distorting the dependency of the last component onto the noise symbol ϵ_1.

Multiple noise symbols. In general, the variable x can be perturbed by multiple noise symbols $\epsilon_1, \epsilon_2, \ldots \epsilon_N$ as

$$\hat{x} = x_0 + \sum_{i=1}^{N} x_i\epsilon_i, \quad \epsilon_i \in [-1, 1],$$

where N is the number of noise symbols for x. The AA model for the floating-point variable x_f is generalized as follows

$$\begin{aligned} \hat{x}_f &= \hat{x}_f^r + \hat{x}_f^e, & (3.23)\\ \hat{x}_f^r &= x_0 + \sum_{i=1}^{N} x_i\epsilon_i, \quad \text{(range component)} & (3.24)\\ \hat{x}_f^e &= B(\hat{x}_f^r) \cdot u \cdot \epsilon_x, \quad \text{(rounding error component)} & (3.25) \end{aligned}$$

where ϵ_x is a new noise symbol representing the uncertainty of the rounding error of the floating-point variable x_f, $\epsilon_x \in [-1, +1]$.

3.3.2 AA-Based Error Models for Basic Floating-Point Operations

Given real variables x, y and z, an unary operator $z = f_1(x)$ and a binary operator $z = f_2(x, y)$, we present here the AA-based models for basic unary and binary floating-point operations including: addition, addition/multiplication with a constant, multiplication, reciprocal, division and square root. Among these operations, the addition and addition/multiplication with a constant are affine operations, while the rest are non-affine operations.

We assume that the range components of two floating-point variables x_f and y_f depend on N noise symbols: $\epsilon_1, \epsilon_2, \ldots, \epsilon_N$. The AA-based models for floating-point variables x_f and y_f are

$$\begin{aligned} \hat{x}_f &= \hat{x}_f^r + \hat{x}_f^e = \hat{x}_f^r + B(\hat{x}_f^r)u\epsilon_x = (x_0 + \sum_{i=1}^{N} x_i\epsilon_i) + B(x_0 + \sum_{i=1}^{N} x_i\epsilon_i)u\epsilon_x, & (3.26)\\ \hat{y}_f &= \hat{y}_f^r + \hat{y}_f^e = \hat{y}_f^r + B(\hat{y}_f^r)u\epsilon_y = (y_0 + \sum_{i=1}^{N} y_i\epsilon_i) + B(y_0 + \sum_{i=1}^{N} y_i\epsilon_i)u\epsilon_y, & (3.27) \end{aligned}$$

where the rounding error components of the two floating-point numbers - $\hat{x}_f^e$ and $\hat{y}_f^e$ - are modeled by the two noise symbols ϵ_x and ϵ_y, respectively. The AA form for the floating-point computed variable z_f is

$$\hat{z}_f = \hat{z}_f^r + \hat{z}_f^e. \tag{3.28}$$

The task here is to determine $\hat{z}_f$ in terms of $\hat{x}_f$ and $\hat{y}_f$. For affine operations, $\hat{z}_f$ can be represented accurately by expanding and rearranging the AA forms of input operands $\hat{x}_f$ and $\hat{y}_f$. For non-affine operations (e.g., multiplication), $\hat{z}_f$ cannot be exactly represented as an affine combination of input operands but has to be approximated, thereby causing the *approximation error* [53].

The range component $\hat{z}_f^r$ and the rounding error component $\hat{z}_f^e$ of the AA form $\hat{z}_f$ for the computed result of binary- and unary-, affine- and non-affine floating-point operations are evaluated by the following expressions:

Addition: $z = x \pm y$

$$\begin{aligned}
\hat{z}_f^r &= \hat{x}_f^r \pm \hat{y}_f^r \\
&= (x_0 \pm y_0) + \sum_{i=1}^{N} (x_i \pm y_i)\epsilon_i \\
\hat{z}_f^e &= \hat{x}_f^e \pm \hat{y}_f^e + B(\hat{z}_f^r)u\epsilon_z \\
&= B(\hat{x}_f^r)u\epsilon_x \pm B(\hat{y}_f^r)u\epsilon_y + B(\hat{z}_f^r)u\epsilon_z
\end{aligned}$$

Addition with a constant: $z = x \pm C$

$$\begin{aligned}
\hat{z}_f^r &= \hat{x}_f^r \pm C \\
&= (x_0 + C) + \sum_{i=1}^{N} x_i\epsilon_i \\
\hat{z}_f^e &= \hat{x}_f^e + B(\hat{z}_f^r)u\epsilon_z \\
&= B(\hat{x}_f^r)u\epsilon_x + B(\hat{z}_f^r)u\epsilon_z
\end{aligned}$$

Multiplication with a constant: $z = Cx$

$$\begin{aligned}
\hat{z}_f^r &= C\hat{x}_f^r \\
&= Cx_0 + \sum_{i=1}^{N} Cx_i\epsilon_i \\
\hat{z}_f^e &= C\hat{x}_f^e + B(\hat{z}_f^r)u\epsilon_z \\
&= CB(\hat{x}_f^r)u\epsilon_x + B(\hat{z}_f^r)u\epsilon_z
\end{aligned}$$

Multiplication: $z = xy$

$$\begin{aligned}
\hat{z}_f^r &= \hat{x}_f^r \cdot \hat{y}_f^r \\
&\approx x_0y_0 + \sum_{i=1}^{N} (x_0y_i + y_0x_i)\epsilon_i + \; B(\sum_{i=1}^{N} x_i\epsilon_i) \cdot B(\sum_{i=1}^{N} y_i\epsilon_i) \cdot \epsilon_{N+1} \\
\hat{z}_f^e &\approx B(\hat{y}_f^r) \cdot \hat{x}_f^e + B(\hat{x}_f^r) \cdot \hat{y}_f^e + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z \\
&\approx B(\hat{y}_f^r) \cdot B(\hat{x}_f^r) \cdot u \cdot \epsilon_x + B(\hat{x}_f^r) \cdot B(\hat{y}_f^r) \cdot u \cdot \epsilon_y + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z
\end{aligned}$$

In the above expressions, C is a constant, $B(\cdot)$ is the hard- or probabilistic bounding operator. Every single floating-point operation generates a rounding error whose size depends on the current working precision (or the unit roundoff u) and the magnitude of the result itself, represented by the term $B(\hat{z}_f^r)\cdot u\cdot\epsilon_z$ and a distinct noise symbol ϵ_z in the error component $\hat{z}_f^e$ in each operation. The rounding error component $\hat{z}_f^e$ includes the rounding error of the current operation, associated with the noise symbol ϵ_z, and the rounding error propagated from the input operands.

For the multiplication $z = xy$, the last term in the range component is an approximation of the product $(\sum_{i=1}^N x_i\epsilon_i)\cdot(\sum_{i=1}^N y_i\epsilon_i)$, as the multiplication is a non-affine operation. The approximation term in the multiplication is associated with a new noise symbol ϵ_{N+1}, which must be distinct from all other noise symbols used so far. It has been shown in [53] that the approximated range for the multiplication, in the worst case, can be four times the true range. Note that for evaluating the rounding error component $\hat{z}_f^e$ of the floating-point multiplication, we skip the term

$$\hat{x}_f^e\cdot\hat{y}_f^e = B(\hat{x}_f^r)u\epsilon_x\cdot B(\hat{y}_f^r)u\epsilon_y = B(\hat{x}_f^r)B(\hat{y}_f^r)\epsilon_x\epsilon_y\cdot u^2,$$

as this term is of order u^2, it is extremely small compared to the other rounding error terms scaled by the unit roundoff u.

The division of two floating-point operands $z = x/y$ is computed via the reciprocal and multiplication as $z = x\times(1/y)$. For the reciprocal $z = 1/y$, which is a non-affine operation, we use the *min-range approximation* [53]. The AA models for the floating-point division and reciprocal are presented in the following.

Division: $z = x/y = x\times(1/y)$

Reciprocal: $z = 1/y$

$$\begin{aligned}
\hat{z}_f^r &= f^{\text{min-range}}\left(1/\hat{y}_f^r\right) && (3.29)\\
&= \alpha\cdot\hat{y}_f^r + C_0 + C_1\cdot\epsilon_{N+1}\\
&= (\alpha\cdot y_0 + C_0) + \sum_{i=1}^N y_i\epsilon_i + C_1\cdot\epsilon_{N+1}\\
\hat{z}_f^e &= B\left(f^{\text{min-range}}\left(1/(\hat{y}_f^r)^2\right)\right)\cdot\hat{y}_f^e + B(\hat{z}_f^r)\cdot u\cdot\epsilon_z && (3.30)\\
&= B\left(f^{\text{min-range}}\left(1/(\hat{y}_f^r)^2\right)\right)\cdot B(\hat{y}_f^r)\cdot u\cdot\epsilon_y + B(\hat{z}_f^r)\cdot u\cdot\epsilon_z,
\end{aligned}$$

where the real coefficients α, C_0 and C_1 are used to approximate a non-affine representation (due to the reciprocal) by an affine form. The coefficient C_1 is called the approximating error associated with a new noise symbol ϵ_{N+1}. For IA and AA, the reciprocal of an interval is defined if that interval does not contain zero. We assume the floating-point variable y_f lies in a positive range $[a, b]$, with $0 < a < b$. Given an AA form for the range component $\hat{y}_f^r = y_0 + \sum_{i=1}^N y_i\epsilon_i$, the lower and upper bounds - a and b - corresponding to the range component can be computed as

$$a = y_0 - \sum_{i=1}^N |y_i|, \quad b = y_0 + \sum_{i=1}^N |y_i|,$$

and the coefficients α, C_0 and C_1 of the min-range approximation $f^{\text{min-range}}\left(1/\hat{y}_f^r\right)$ of a reciprocal $1/\hat{y}_f^r$ are evaluated as follows [53]

$$\begin{aligned}
\alpha &= -1/b^2 \\
d_1 &= (1/a) - \alpha \cdot a \\
d_2 &= (1/b) - \alpha \cdot b \\
C_0 &= (d_1 + d_2)/2 \\
C_1 &= (d_1 - d_2)/2.
\end{aligned}$$

For the case of a negative range, i.e., $a < b < 0$, we need to take the absolute values of the bounds, i.e., $a = |a|$ and $b = |b|$, and then swap them before performing the same calculations as above (to make sure b is always the larger coefficient in absolute value); additionally, after the calculation of C_0, its sign needs to be inverted.

Similarly, the AA form for the rounding error component $\hat{z}_f^e$ of a floating-point reciprocal is approximated by using the min-range approximation. Be noted that the min-range approximation is applied to the inverse of the square of range component, then the bounding operator is applied. The detailed derivation is presented in Appendix A. Similar to the other floating-point operations, a new noise symbol ϵ_z associated with the last error term of $\hat{z}_f^e$ is introduced to represent the rounding error of the current reciprocal operation.

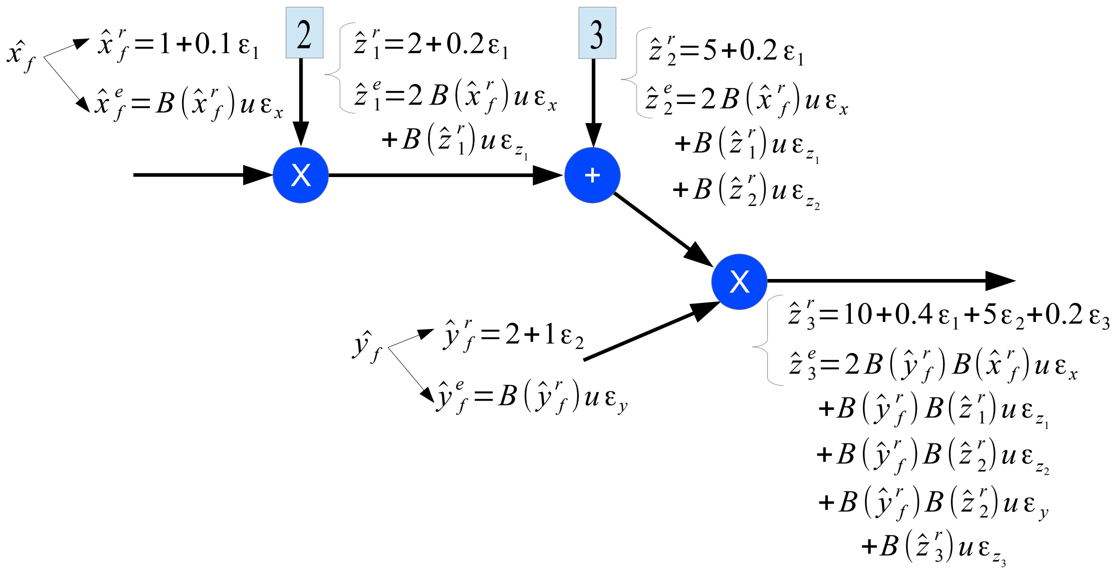
Square root: $z = \sqrt{x} = sqrt(x)$

$$\begin{aligned}
\hat{z}_f^r &= f^{\text{Chebyshev}}\left(\sqrt{\hat{x}_f^r}\right) \\
&= \alpha \cdot \hat{x}_f^r + C_0 + C_1 \cdot \epsilon_{N+1} \\
&= (\alpha \cdot x_0 + C_0) + \sum_{i=1}^{N} x_i \epsilon_i + C_1 \cdot \epsilon_{N+1} \\
\hat{z}_f^e &= \frac{1}{2} B\left(f^{\text{min-range}}\left(1/\hat{z}_f^r\right)\right) \cdot \hat{x}_f^e + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z. \\
&= \frac{1}{2} B\left(f^{\text{min-range}}\left(1/\hat{z}_f^r\right)\right) \cdot B(\hat{x}_f^r) \cdot u \cdot \epsilon_x + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z.
\end{aligned}$$

We follow the Chebyshev approximation presented in [53] to approximate the range component $\hat{z}_f^r$ of the AA model for a floating-point square root $z = \sqrt{x}$. The rounding error component $\hat{z}_f^e$ is approximated by using the min-range approximation, as used before by the AA model for floating-point reciprocal. The detailed derivation for the rounding error component $\hat{z}_f^e$ of the floating-point square root is presented in Appendix B. We summarise here only the main expressions.

For the case of a square root, the real coefficients α, C_0 and C_1 have similar meanings as for the case of a reciprocal. However, the evaluation of these coefficients is different for the square root. The square root is defined for non-negative numbers only. As $\sqrt{0} = 0$, we only consider the cases of positive ranges. We assume the floating-point variable x_f lies in a positive range $[a, b]$, with $0 < a < b$. Given the range component $\hat{x}_f^r = x_0 + \sum_{i=1}^{N} x_i \epsilon_i$, the lower and upper bounds corresponding this range component are

$$a = x_0 - \sum_{i=1}^{N} |x_i|, \quad b = x_0 + \sum_{i=1}^{N} |x_i|,$$



$\hat{x}_f$
$\hat{x}^r_f = 1 + 0.1\varepsilon_1$
$\hat{x}^e_f = B(\hat{x}^r_f)u\varepsilon_x$
2
$\hat{z}^r_1 = 2 + 0.2\varepsilon_1$
$\hat{z}^e_1 = 2B(\hat{x}^r_f)u\varepsilon_x + B(\hat{z}^r_1)u\varepsilon_{z_1}$
X
3
+
$\hat{z}^r_2 = 5 + 0.2\varepsilon_1$
$\hat{z}^e_2 = 2B(\hat{x}^r_f)u\varepsilon_x + B(\hat{z}^r_1)u\varepsilon_{z_1} + B(\hat{z}^r_2)u\varepsilon_{z_2}$
X
$\hat{y}_f$
$\hat{y}^r_f = 2 + 1\varepsilon_2$
$\hat{y}^e_f = B(\hat{y}^r_f)u\varepsilon_y$
$\hat{z}^r_3 = 10 + 0.4\varepsilon_1 + 5\varepsilon_2 + 0.2\varepsilon_3$
$\hat{z}^e_3 = 2B(\hat{y}^r_f)B(\hat{x}^r_f)u\varepsilon_x + B(\hat{y}^r_f)B(\hat{z}^r_1)u\varepsilon_{z_1} + B(\hat{y}^r_f)B(\hat{z}^r_2)u\varepsilon_{z_2} + B(\hat{y}^r_f)B(\hat{z}^r_2)u\varepsilon_y + B(\hat{z}^r_3)u\varepsilon_{z_3}$

Figure 3.3 AA-based evaluation of a sequence of floating-point operations for computing $z = (2x+3)y$

and the coefficients α, C_0 and C_1 of the Chebyshev approximation $f^{\text{Chebyshev}}\left(\sqrt{\hat{x}^r_f}\right)$ of the square root are evaluated as follows [53]

$$\begin{aligned}
\alpha &= 1/(\sqrt{a}+\sqrt{b}) \\
d_1 &= (\sqrt{a}+\sqrt{b})/4 \\
d_2 &= \sqrt{ab}/(\sqrt{a}+\sqrt{b}) \\
C_0 &= (d_1+d_2)/2 \\
C_1 &= (d_1-d_2)/2.
\end{aligned}$$

Error propagation in a sequence of floating-point operations

For a sequence of floating-point operations, the rounding error is propagated through that sequence, described by the example shown in Figure 3.3. This example demonstrates the AA-based evaluation of a sequence of floating-point operations for computing $z = (2x+3)y$, which can be split into three sequential operations:

$$z_1 = 2x, \quad z_2 = z_1 + 3, \quad z = z_3 = z_2 y.$$

We assume the input operands x and y have the AA forms as follows

$$\begin{aligned}
\hat{x}_f &= \hat{x}^r_f + \hat{x}^e_f = (1+0.1\epsilon_1) + B(\hat{x}^r_f)u\epsilon_x \\
\hat{y}_f &= \hat{y}^r_f + \hat{y}^e_f = (2+1\epsilon_2) + B(\hat{y}^r_f)u\epsilon_y.
\end{aligned}$$

We would like to demonstrate how the rounding errors in a sequence of operations are propagated. We, therefore, express the rounding error components in a more generic representation during the computational sequence, while the range components are explicitly evaluated. The evaluated result is shown in Figure 3.3. We assume a hard bounding operator B_{hard} is used and the unit roundoff u can be generic. The final result is explicitly described as

$$\begin{aligned}\hat{z}_f^r &= 10 + 0.4\epsilon_1 + 5\epsilon_2 + 0.2\epsilon_3,\\ \hat{z}_f^e &= 6.6u\epsilon_x + 6.6u\epsilon_{z_1} + 15.6u\epsilon_{z_2} + 15.6u\epsilon_y + 15.6u\epsilon_{z_3}.\end{aligned}$$

For the range components, the evaluation of the multiplication, i.e., a non-affine operation, generates a new noise symbol ϵ_3 added to the final AA form, while the evaluations of the other affine operations do not. For the rounding error components, at least one new noise symbol, associated with the range component of the current computed result, is generated after the evaluation of each operation to represent the rounding error of that operation, regardless of using non-affine or affine operations. For non-affine operations, the rounding errors of the inputs are propagated and scaled over those operations. All the rounding errors are captured and propagated along the entire computational chain allowing for an accurate estimate of the final rounding error.

3.4 AA-Based Error Model for a Fused Multiply-Accumulate

3.4.1 Motivation

The fused multiply-accumulate (FMA) floating-point operation performs a multiplication followed by an addition $z = xy + w$ within one atomic operation and with one single rounding. According to Higham [5], an FMA performs only one rounding and therefore has the conventional error model as follows

$$\hat{z} = fl(xy + w) = (xy + w)(1 + \delta), \quad |\delta| \leq u, \tag{3.31}$$

which implies that $(xy + w)$ is computed exactly and then rounded to the current working precision.

In general, FMA makes it possible to achieve, thanks to its superior benefits, more accurate numerical results and better performance[3] for many applications including dot products, matrix multiplications and polynomial evaluations. Linear transforms can be efficiently mapped to FMAs as showed in [55]. FMA is particularly very useful in the compensated floating-point summation [5] where the rounding error is estimated along with the evaluation of the algorithm at hand and then used as a correction term later to reduce the overall rounding error. Ogita *et al.* [56] suggested the *error free transformations* (EFTs) based on the use of FMA for accurate evaluations of the rounding errors of floating-point multiplication and addition at the same working precision. Similarly, the rounding error of an FMA can be computed exactly by using EFTs, studied by Boldo and Muller [57, 58]. Those findings allow for implementations of EFT-based compensated algorithms for accurate floating-point dot-product and polynomial evaluation [59, 60]. FMA is therefore a desirable functional logic in floating-point units [61, 62].

However, existing work simply uses the conventional error model of FMA as shown in (3.31). To our knowledge, there exists no other suggestion for an error model of FMA using AA in the literature. Due to the importance of FMA, we suggest the first AA-based error model for FMA. We present the model in detail in the following.

[3]This is achievable only if the FMA is as fast as an addition or multiplication separately, and the application at hand needs to have reasonable balanced addtions/multiplications.

3.4.2 AA-Based Error Model for FMA

Basically, an FMA is a combination of an exact multiplication (i.e., without rounding) followed by an addition; the two operations are performed with only one final rounding. Therefore, our suggestion for an AA error model for the FMA is as follows. We combine the AA model of a floating-point multiplication with the AA model of a floating-point addition in a sequential manner, *without introducing the rounding error term in the multiplying step*, to obtain the AA model for a floating-point FMA. Details for obtaining the AA model of an FMA are shown in two steps by the following equations:

$$z \;=\; f_3(x,y,w) = xy + w = v + w, \quad v = xy.$$

Step 1. Multiplication without rounding: $v = xy$

$$\begin{aligned} \hat{v}_f^r &= x_0y_0 + \sum_{i=1}^{N}(x_0y_i + y_0x_i)\epsilon_i + B(\sum_{i=1}^{N} x_i\epsilon_i)\cdot B(\sum_{i=1}^{M} y_i\epsilon_i)\cdot\epsilon_t, \\ \hat{v}_f^e &= B(\hat{x}_f^r)\cdot\hat{y}_f^e + B(\hat{y}_f^r)\cdot\hat{x}_f^e, \end{aligned}$$

Step 2. Addition: $z = v + w$

$$\begin{aligned} \hat{z}_f^r &= \hat{v}_f^r + \hat{w}_f^r, \\ \hat{z}_f^e &= \hat{v}_f^e + \hat{w}_f^e + B(\hat{z}_f^r)\cdot u\cdot\epsilon_{k_z}. \end{aligned}$$

Finally, the AA form for $z = xy + w$ is

$$\hat{z}_f \;=\; \hat{z}_f^r + \hat{z}_f^e.$$

3.5 Estimation of Rounding Error Bound from AA Form

The range component $\hat{z}_f^r$ and error component $\hat{z}_f^e$ of an affine form $\hat{z}_f$ modeling the corresponding range and rounding error of a floating-point computed result $z = f_1(x)$, $z = f_2(x,y)$ of the basic operations, or $z = f(x,y,w)$ of a fused operation can be used to derive the respective numerical range and rounding error bound by applying the bounding operator. We are interested in the rounding error and apply the bounding operator $B(\cdot)$ on the error component $\hat{z}_f^e$ for estimating the rounding error bound of variable z as follows

$$E_{max,z} \;=\; B(\hat{z}_f^e), \tag{3.32}$$

where $E_{max,z}$ is the estimated rounding error bound.

3.6 AA-Based Error Model versus Conventional Error Model

The previous sections have described how AA is used to model floating-point arithmetic. Here we wish to know the differences between the conventional error model (subsection 3.2.1) and the AA-based error model (subsection 3.3.1) in representation and evaluation of floating-point arithmetic.

For the representation of a floating-point number, the AA-based error model in (3.17) is equivalent to the conventional error model in (3.1), in which the relative error δ is replaced by the unit roundoff u and the noise symbol ϵ.

However, for the evaluation of floating-point arithmetic operations, the AA-based error model is different from the conventional error model. The conventional error model in equation (3.4) uses a symbol δ to evaluate the (relative) rounding error of a single floating-point operation. For a sequence of floating-point operations, the conventional error model first starts with different symbols $\delta_1, \delta_2, \ldots, \delta_n$ for the rounding errors of different operations in that sequence. Then for simplicity, it is assumed that all these symbols are identical and therefore one sole symbol δ is necessary for the representation and evaluation of both a single floating-point number/operation and a sequence of floating-point operations. This assumption often leads to a final error expression consisting of high order terms in δ, i.e., some power of δ depending on the complexity of the algorithm and the problem size. As a consequence, the rounding error bound derived by the conventional error model is very conservative and often depends on one or several orders of the problem size, making conventional error analysis unsuitable for bit width allocation.

In AA-based error modeling, each rounding error term in a sequence of floating-point operations is represented by a distinct noise symbol ϵ_i, and all the rounding error terms are captured, potentially offering error cancelation. The final AA-based rounding error expression is a first-order polynomial of all noise symbols allowing for the use of the central limit theorem and the probabilistic bounding operator (subsection 3.2.3). The error bound estimated by applying an AA-based probabilistic bounding operator is, therefore, much closer to the real errors in practical applications (which is presented in more detail in Chapter 5), making AA-based error analysis a practical approach for bit width allocation. The downside of AA-based error analysis is that it is more expensive to do.

3.7 Conclusions

This chapter has presented the fundamentals of floating-point error analysis using affine arithmetic. The core theory is based on AA error models for floating-point numbers and floating-point arithmetic, allowing for the representation and evaluation of floating-point computations with a hard- or probabilistic bounding operator. In this chapter, we also suggested an AA-based error model for a fused multiply-accumulate operation $z = xy + w$, which, to our knowledge, is the first time AA is used to model a floating-point FMA.

These fundamentals will be the basis for building a Matlab-based framework presented in Chapter 4, and they are necessary for the automation of floating-point error analysis and bit width allocation of floating-point signal and speech processing algorithms, presented later in Chapter 5.

4 Floating-Point Error Analysis Using Affine Arithmetic: A Matlab-based Framework

4.1 Introduction

4.1.1 Motivation

One main goal of this work is to perform the floating-point rounding error analyses of signal processing algorithms and iterative algorithms in real world applications. Therefore, usability and scalability are two desirable characteristics of a software tool used to perform floating-point rounding error analysis.

Using the AA-based error model for floating-point arithmetic presented in Chapter 3, we implement a Matlab-based framework to model and estimate the rounding error bound for arbitrary floating-point algorithms. Matlab was chosen because of its widespread use in many areas including signal processing and scientific computing.

The basic idea behind our Matlab-based tool is to define a new class in Matlab, the `AAFloat` class, for representing affine forms of floating-point operands and executing floating-point operations. Our motivation of implementing a Matlab class rather than a set of functions for AA-based error modeling is to enable users to reuse their existing Matlab code with minimal modifications. Our implemented framework supports commonly-used floating-point operations, including $\{+, -, *, /, \}$, square root (sqrt), and fused multiply-add, and is able to fully operate with vector and matrix operations of Matlab. These features allow for best reuse of existing Matlab codes (with some minor changes) to effectively conduct rounding error estimation for floating-point algorithms.

4.1.2 Related Software Tools

Table 4.1 presents a comparison between this thesis and other existing work performing finite-precision error analysis using AA.

Fang *et al.* [4, 28, 42] employed affine arithmetic to model rounding error of fixed-point and floating-point implementations of linear transform algorithms. The key idea applied in Fang's work is to use an AA-based probabilistic bounding operator to obtain a reliable estimate of the final rounding error, thereby allowing for deriving an optimal uniform bit with configuration. The tool Fang built is automatic and introduces probabilistic bounds, but it is not integrated with Matlab. Also, linear transforms require only additions and multiplications by constants.

Table 4.1 Existing work on AA-based finite-precision error analysis

	Fang [4, 28, 42]	MiniBit [24]	MiniBit+ [25]	**This thesis**
Number format	Fixed-point Floating-point	Fixed-point	Floating-point	Floating-point
Bit width configuration	Uniform	Multiple	Multiple	Uniform
Supported arithmetic	$+, -, \times,$ /	$+, -, \times$	$+, -, \times$	$+, -, \times,$ /, `fma`, `sqrt`
Bounding operator	Hard, Probabilistic	Hard	Hard	Hard, Probabilistic
Language	C++	C++ (& ASA)	C++ (& ASA)	Matlab
Custom-precision FP library	SystemC, CMU*float*	-	-	MPFR
Example given	FIR25, WHT64, DCT8, IDCT8, IIR (2nd-order), Cholesky (range estimation only).	Polynomial 4, `2x2` Matrix Mul., RGB to YCbCr, B-Splines, DCT8.	RGB to YCbCr, B-Splines, DCT8.	General dot products with vector length up to 1000, Levinson-Durbin (range + error estimation).
Max. number of error terms in examples	< 500	< 100	< 100	≈ 7000

Lee *et al.* [24] presented MiniBit, an automatic and static approach for optimizing bit width of fixed-point feedforward designs, which was based on AA combined with adaptive simulated annealing for obtaining a multiple fraction bit width configuration for fixed-point arithmetic. In contrast to Fang's work, MiniBit estimates the worst case error of fixed-point algorithms. MiniBit was later extended to floating-point arithmetic in MiniBit+ [25].

The existing work of Fang, MiniBit and MiniBit+ were implemented using C/C++. The usage of those software tools, however, was described quite briefly in the literature. These facts expose limitations to common users who may want to have an ease-of-use software tool in order to quickly perform error analyses for their own numerical algorithms with very little effort. It is also not easy for Matlab users to transfer their existing `.m` scripts into C/C++ codes in order to run the desired error analysis programs.

This motivates us to implement our software tool that allows for an accurate and convenient estimation of the rounding errors of floating-point algorithms in the Matlab environment, and to our knowledge, it is the first Matlab tool to perform floating-point error analysis based on AA models.

4.1.3 Framework Overview

A visualization of the proposed Matlab-based framework is presented in Figure 4.1. The framework consists of two main tasks: Error Bound Estimation and Error Verification, displayed on the left and right paths of Figure 4.1, respectively. The Matlab framework supports both tasks.

The *Error Bound Estimation* task is performed by using the `AAFloat` class in Matlab. Given a floating-point algorithm that consists of arithmetic expressions performed at a working precision p, all floating-point variables of the algorithm are first converted into `AAFloat` objects, which are AA forms representing respective floating-point operands. Then all floating-point expressions are

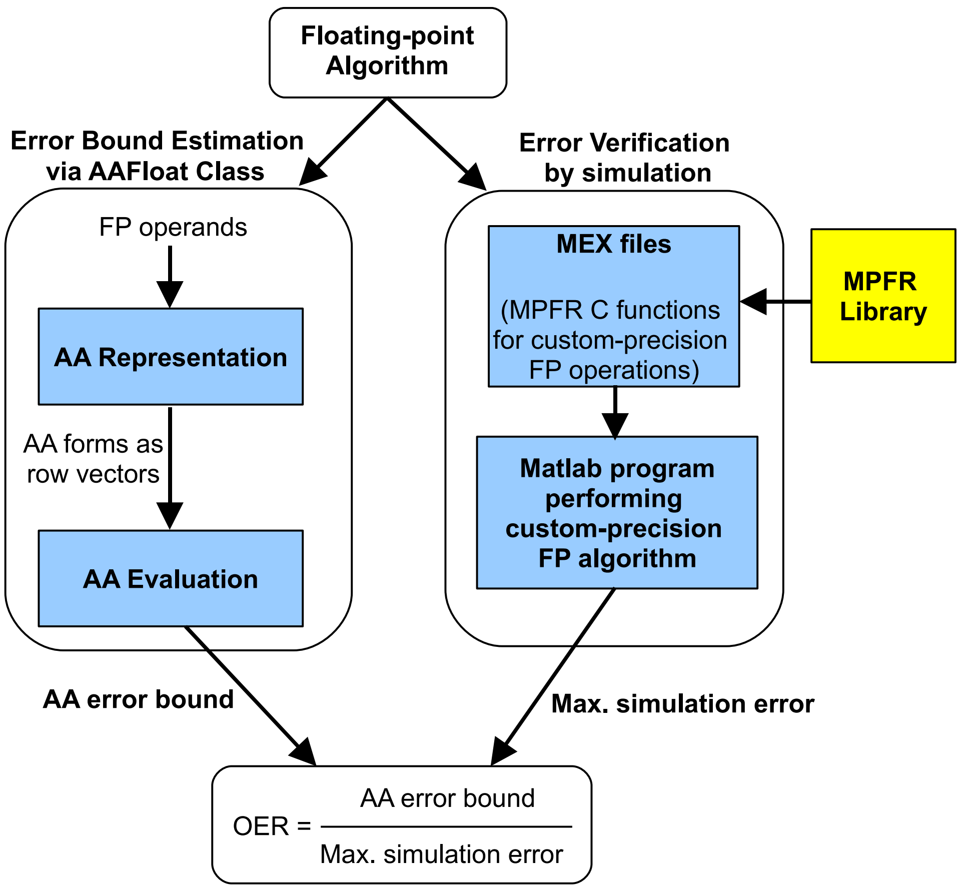

Floating-point Algorithm
Error Bound Estimation via AAFloat Class
Error Verification by simulation
FP operands
AA Representation
AA forms as row vectors
AA Evaluation
MEX files
(MPFR C functions for custom-precision FP operations)
MPFR Library
Matlab program performing custom-precision FP algorithm
AA error bound
Max. simulation error
OER = AA error bound / Max. simulation error

Figure 4.1 Matlab-based framework for AA-based floating-point rounding error evaluation and verification

evaluated following corresponding affine arithmetic expressions, and the AA error bounds of the desired computational quantities are estimated. Generally, the user does not need to change his/her Matlab code to use the tool.

For *Error Verification* task, we conduct extensive simulations on a sufficiently large set of testing samples in order to determine the maximum rounding error of the algorithm at working precision p.

Over-Estimation Ratio: We define the *Over-Estimation-Ratio* (OER)

$$OER = \frac{\texttt{AA error bound}}{\texttt{Maximum simulation error}}. \tag{4.1}$$

as the ratio of the estimated AA-based rounding error bound over the actual maximum simulation error. We use this ratio to evaluate the tightness of the bound and the reliability of the AA-based error analysis technique. Ideally, an OER of 1.0 is desirable, meaning that the bound is tight, but this is in general not achievable. In practice, one hopes for an OER larger than 1.0 but very close to 1.0. In this case the AA-based rounding error bound can be seen as a reliable estimate.

In the remaining chapter, details on the implementations of the Error Estimation task and the Error Verification task in Figure 4.1 are presented.

4.2 Error Estimation using The AAFloat Class

4.2.1 AAFloat Overview

We present in this section the `AAFloat` class in Matlab, corresponding to the left part in Fig. 4.1, to implement AA error models for floating-point arithmetic.

With the `AAFloat` class, from an existing code (.m file) executing an algorithm in Matlab, users can easily obtain another code that models the floating-point rounding error of some computational quantities by replacing floating-point computational operations in the original code with respective Matlab operators and methods supported by the class to evaluate those floating-point operations following affine arithmetic. Due to the operator overloading feature by using the `AAFloat` class in Matlab, the user in general only needs to make minimal modifications in the original Matlab code. The rounding error analysis task, therefore, can be performed effectively from an existing .m file in the same Matlab working environment, which typically allows for high productivity in comparison with other C-based software.

```
%----------------------------
% original code: dot-product
%----------------------------
%
%
%
s = x(1) * y(1);
for i=2:n
     s = s + x(i) * y(i);
end;
```

```
%------------------------------------------
% AA-based error analysis with AAFloat
%------------------------------------------
AAFloat_INIT;
x = AAFloat(-a*ones(n,1), a*ones(n,1), p);
y = AAFloat(-b*ones(n,1), b*ones(n,1), p);
s = x(1) * y(1);
for i=2:n
     s = s + x(i) * y(i);
end;
E_BOUND_s = ebound(s);
```

The two segments of code above briefly show how the original Matlab code for computing a sequential dot-product is reused for the AA-based rounding error evaluation with the `AAFloat` class. Details on the implementation of methods and functions in the `AAFloat` class are presented later.

Terms and Notations

In the following, we will describe the implementation of the `AAFloat` class in Matlab for conducting AA-based error estimation. We will use the terms "class" or "tool" or "software tool" to refer to our `AAFloat` class implemented in this work.

We follow Matlab notation when presenting `AAFloat` class methods and examples. The upper case symbols like `X`, `Y`, `A`, `B`, etc. denote `double` matrices in Matlab, while lower case symbols like `x`, `y`, etc. denote `double` vectors or scalars. The upper case symbols suffixed by `_AA` like `X_AA`, `Y_AA`, etc. denote `AAFloat` matrices, whose elements are `AAFloat` objects representing respective floating-point operands. Similarly, the lower case symbols suffixed by `_AA` like `x_AA`, `y_AA`, etc. denote `AAFloat` vectors or scalars. The working precision of floating-point numbers is denoted with `p`, which is a scalar integer value.

Representation of floating-point operand in `AAFloat` class

The first implementation issue is to effectively represent the input floating-point operands as input AA forms that consist of the range components and error components (see Section 3.3). Affine forms can become very complex expressions after multiple arithmetic operations have been applied to the input form. In that context, the use of symbolic computation is one possible choice to represent the affine forms, which however potentially makes the software tool for affine evaluation very costly with respect to memory usage and execution time.

In contrast to a symbolic framework, we represent AA models of floating-point operands as numerical row vectors in the `AAFloat` class. More specifically, each affine form corresponding to one floating-point operand is represented in Matlab workspace by two row vectors: one for the range component and the other one for the rounding error component. Additionally, we use one extra scalar to store the working precision of the floating-point operand.

In short, each floating-point operand `x` (a scalar) is represented by one instance `x_AA` (object) of the `AAFloat` class, whose class properties are two row vectors `x_AA.r` and `x_AA.e` for representing range- and error components, and one scalar `x_AA.p` for storing precision. Each element of the two row vectors is used to store either the central value or the partial deviation associated with respective noise symbol ϵ_i. For the range component, `x_AA.r(1)` stores the central value while entry `x_AA.r(i+1)` stores the coefficient for the *ith* noise term ϵ_i. Similarly, for the rounding error component, the entry `x_AA.e(i)` stores the coefficient for the *ith* noise term.

For example, a floating-point variable `x` represented at single-precision ($p = 24$) and having the affine form $\hat{x} = 1 + 2\epsilon_1 - 3\epsilon_2 + 0.5\epsilon_3$ is generated and represented in Matlab with the `AAFloat` class as follows:

```
>> AAFloat_INIT
>> x_AA = AAFloat([1 2 -3 0.5], 24)
x_AA =
AAFloat matrix of size [1 x 1]
(1,1)
```

```
r: [1 2 -3 0.5]
e: 3.8743e-07
p: 24
```

4.2.2 AAFloat Methods

Table 4.2 lists the implemented methods for the `AAFloat` class. One prominent feature of the `AAFloat` class is that it allows for vectorization of floating-point computations following AA models. It automatically works for vectors and matrices in Matlab. This means that vectors and matrices of `AAFloat` objects can be generated in the Matlab workspace by one single call to the class constructor method, and floating-point arithmetic operations accept input operands as Matlab vectors and matrices. With this feature, the transfer from the existing .m code executing a floating-point application into the .m code performing floating-point error estimation for the same application can be carried out with only some minor changes in the original code, most of the code for algorithmic computations is reusable. The following describes in detail the features of the `AAFloat` class.

Initialization of the `AAFloat` class

The initialization of the `AAFloat` class *must* be performed with the function `AAFloat_INIT` before any call to other methods of the class. This initialization function creates in Matlab workspace one global variable `AAFloat_MAX_INDEX` used to capture the maximum number of noise terms occurring in all existing affine forms.

`AAFloat_MAX_INDEX` has two fields `AAFloat_MAX_INDEX.nR` and `AAFloat_MAX_INDEX.nE` for storing and updating the maximum number of error terms in the range component and (rounding) error component of all affine forms, respectively. Because new noise terms can continuously be generated during the creation of a new affine form or the evaluation of a floating-point operation based on AA, the values of fields `.nR` and `.nE` are, therefore, continuously updated within those operations in order to guarantee that the maximum numbers of noise terms in range- and error components are accurately tracked. Once initialized, the fields `.nR` and `.nE` of `AAFloat_MAX_INDEX` are set to zero.

Class constructor method

The class constructor method `AAFloat` is used to generate an `AAFloat` matrix from specified numerical ranges of floating-point input operands and respective working precision. Note, that `AAFloat` vector and scalar are special cases of `AAFloat` matrix. The `AAFloat` matrix is a Matlab matrix of which each element is one `AAFloat` object that models one floating-point scalar variable. We implemented a variety of syntax variants for the `AAFloat` constructor method (see Table 4.2) as described below.

1. `X_AA = AAFloat(A,B,p)` generates an `AAFloat` matrix to model a floating-point matrix `X` using affine arithmetic. The numerical ranges of the floating-point matrix `X` being modeled are defined by input matrices `A` and `B`. Matrix `A` defines the lower bounds and matrix `B` defines the upper bounds for the intervals of the elements of `X`, for which the numerical range of element `X(m,n)` is the interval specified by `[A(m,n), B(m,n)]`, where `m` and `n` are matrix indices. Matrices `A` and `B` must have the same size and, therefore, `X_AA` has the same size as `A` and `B`. The scalar `p` defines the working precision of `X` (see Equ. (3.3)) and must be an integer.[1]

[1]As this work performs uniform bit width allocation, all floating-point variables and operations are assume to have the same precision p.

Table 4.2 List of methods for AA-based evaluation of floating-point operations via the `AAFloat` class

Name	Matlab syntax	Description
`AAFloat_INIT`	`AAFloat_INIT`	Initialization of `AAFloat`

`AAFloat`	`X_AA = AAFloat(A,B,p)`
	`X_AA = AAFloat(A,B)`
	`X_AA = AAFloat(a,p)`
	`X_AA = AAFloat`
		Class constructor

`add`	`Z_AA = add (X_AA,Y_AA,prob)`	addition
`sub`	`Z_AA = sub (X_AA,Y_AA,prob)`	subtraction
`mmul`	`Z_AA = mmul(X_AA,Y_AA,prob)`	matrix multiplication
`mul`	`Z_AA = mul (X_AA,Y_AA,prob)`	element-wise multiplication
`inv`	`Z_AA = inv (Y_AA,prob,userrange)`	element-wise reciprocal
`div`	`Z_AA = div (X_AA,Y_AA,prob,userrange)`	element-wise division
`sqrt`	`Z_AA = sqrt(X_AA,prob,userrange)`	element-wise square root
`fma`	`Z_AA = fma (X_AA,Y_AA,W_AA,prob)`	element-wise fused multiply-add
`sum`	`Z_AA = sum (X_AA,prob)`	summation
`plus`	`Z_AA = X_AA + Y_AA`	`add` with default bound
`minus`	`Z_AA = X_AA - Y_AA`	`sub` with default bound
`mtimes`	`Z_AA = X_AA * Y_AA`	`mmul` with default bound
`times`	`Z_AA = X_AA .* Y_AA`	`mul` with default bound
`rdivide`	`Z_AA = X_AA ./ Y_AA`	`div` with default bound
`ebound`	`ERR_BOUND = ebound(Z_AA,prob)`	estimate error bound
`rbound`	`RANGE_BOUND = rbound(Z_AA,prob)`	estimate range bound

The parameter `prob` specifies the type and confidence interval of the bound estimated within the evaluating function: a hard bound corresponds to `prob` = 1; a probabilistic bound corresponds to 0 < `prob` < 1; the default setting is `prob` = 0.9973 (99.73% or 3σ, see 3.2.3). The parameter `userrange` specifies the desired range of the user for handling special cases, i.e., division by zero and square root of a negative range; see subsection 4.2.2.

2. `X_AA = AAFloat(A,B)` generates an `AAFloat` matrix to model a floating-point matrix `X` in single-precision number format (`p=24`) using affine arithmetic. Matrices `A` and `B` have the same meaning as described in the first syntax variant of the `AAFloat` constructor method.

3. `X_AA = AAFloat(a,p)` generates an `AAFloat` object to model a floating-point variable `x` in precision p using affine arithmetic. The entries of input vector `a` specify elements of the range component of the respective AA form, i.e., the central value `x_AA.r[1]` corresponds to `a[1]` and other terms `x_AA.r[i]` correspond to `a[i] (i = 2, 3, etc.)`[2].

4. `X_AA = AAFloat`, with no input argument, generates one `AAFloat` object that models a floating-point variable in the range `[-1,+1]` in single-precision format (`p=24`).

In all the syntax variants for the `AAFloat` constructor method above, users need to specify the range component `x_AA.e` and the precision `p` only. The rounding error component `x_AA.e` is implicitly estimated from the value of range component and the precision using the bounding operator $B(\cdot)$, as shown by Equ. (3.25).

Example 4.1. The following is an example of using the `AAFloat` constructor method to generate an `AAFloat` matrix `X_AA` to represent a single-precision floating-point 2-by-2 matrix `X` having four elements: `X(1,1)` $\in [-1, 1]$, `X(1,2)` $\in [-2, 2]$, `X(2,1)` $\in [-3, 3]$ and `X(2,2)` $\in [-4, 4]$.

```
>> format short
>> AAFloat_INIT
>> X_AA = AAFloat ([-1,-2; -3,-4], [1,2; 3,4], 24)
X_AA =
AAFloat matrix of size [2 x 2]
(1,1)
    r: [0 1]
    e: 5.9605e-08
    p: 24
(1,2)
    r: [0 0 2]
    e: [0 1.1921e-07]
    p: 24
(2,1)
    r: [0 0 0 3]
    e: [0 0 1.7881e-07]
    p: 24
(2,2)
    r: [0 0 0 0 4]
    e: [0 0 0 2.3842e-07]
    p: 24
```

Example 4.2. The second example generates an `AAFloat` column vector `y_AA` of size 2-by-1 to represent a single-precision floating-point vector `y` having two elements: `y(1,1)` $\in [-5, 10]$ and `y(2,1)` $\in [3, 7]$.

```
>> y_AA = AAFloat ([-5; 3], [10; 7])
y_AA =
AAFloat matrix of size [2 x 1]
(1,1)
```

[2]Note that the index in Matlab starts from 1 while the index in the analytical AA model in (3.23 - 3.25) starts from 0, therefore the central value x_0 of AA form corresponds to `a[1]` and `x_AA.r[1]` in Matlab.

```
        r: [2.5000 7.5000]
        e: 5.9605e-07
        p: 24
    (2,1)
        r: [5 0 2]
        e: [0 4.1723e-07]
        p: 24
```

Example 4.3. This example generates an `AAFloat` object `x_AA` to represent a floating-point variable `x` represented at precision p=30 and having the affine form $\hat{x} = 1 + 2\epsilon_1 + 0\epsilon_2 - 3\epsilon_3 - 1\epsilon_4$.

```
>> x_AA = AAFloat ([1, 2, 0, -3,-1], 30)
x_AA =
AAFloat matrix of size [1 x 1]
(1,1)
    r: [1 2 0 -3 -1]
    e: 6.5193e-09
    p: 30
```

Vectorization of floating-point operations

The `AAFloat` class allows for an easy AA-based evaluation of vectorized floating-point operations. All implemented operations within the `AAFloat` class can operate with inputs `X_AA`, `Y_AA` and `W_AA` as vectors or matrices of the `AAFloat` class or `double` class provided that at least one input operand belongs to the `AAFloat` class.

The implemented operations can be divided into two groups: the basic operation group and specialised operation group. The basic floating-point operation group includes addition (`add`), subtraction (`sub`), matrix multiplication (`mmul`), element-wise multiplication (`mul`), reciprocal (`inv`), division (`div`) and square root (`sqrt`), and their respective Matlab operator overloading functions, i.e., `plus`, `minus`, `mtimes`, `times`, and `rdivide`, respectively. Operator overloading functions allow for a convenient usability of standard Matlab operators $\{+, -, *, .*, ./\}$. The reciprocal, division and square root operations perform in an element-wise manner only. The division is executed indirectly via reciprocal and multiplication (cf. section 3.3.2).

The specialised operation group consists of element-wise fused multiply-add (`fma`) and summation (`sum`). The `fma` calculates the combination of one multiplication followed by one addition with only one final rounding, thereby potentially offering a more accurate computed result. Similar to the `inv` and `div` operations, the `fma` operates in an element-wise manner, i.e., `Z_AA(m,n) = X_AA(m,n)*Y_AA(m,n) + W_AA(m,n)` is executed with one rounding.

The `sum` overloads the standard sum function of Matlab. If `X_AA` is a vector of `AAFloat` objects, `sum` returns the sum of all elements. If `X_AA` is a matrix of `AAFloat` objects, `sum` treats each column as vector, returning a row vector of the sums of each column.

Matrix dimension agreement and constant input

In general, the operations supported by the `AAFloat` class require that input matrices, no matter whether they belong to either the `AAFloat` class or `double` class, must agree on dimensions. For addition, subtraction, element-wise multiplication, division and fused multiply-add, all input matrices must have the same size. For matrix multiplication, the number of columns of the first matrix has to be equal to the number of rows of the second matrix.

There is also an exception, when one input argument is a scalar (of `AAFloat` class or `double` class), the respective operation will be executed in an element-wise manner, i.e., the scalar is added, subtracted, multiplied, or divided with each element of the other input argument.

The `AAFloat` class allows for a mixture of `AAFloat` and `double` classes as inputs, i.e., input arguments can be matrices, vectors or scalars in `double`. The only requirement is that for each call of an `AAFloat` method at least one input argument must be an `AAFloat` object. If an input argument belongs to `double` class, it will first be internally converted into an `AAFloat` object before being used for evaluating the respective floating-point operation following affine arithmetic.

Bounding operator

The parameter `prob` (i.e., short for *probability*) is used to specify the type and confidence interval of the bound which will be estimated within the evaluating function (cf. Section 3.3.2). The value of `prob` must be in the range $(0, 1]$. A hard bound corresponds to `prob` $= 1$ or a probability of 100%. Any other assigned probability less than 1 will implicitly specify a probabilistic bounding operator to be used. The default setting is `prob` $= 0.9973$, which means to use a probabilistic bounding operator with a confidence interval of exactly three times ($K = 3$) the standard deviation 3σ (or about 99.73% of confidence, also cf. Section 3.2.3).

Matlab operator overloading

For those cases that apply the default setting of the bounding operator (i.e., a probabilistic bound), basic floating-point operations can simply be executed by overloading with standard Matlab operators. Operator overloading of Matlab functions is available within the `AAFloat` class via our definitions of standard Matlab methods {`plus`, `minus`, `mtimes`, `times`, `rdivide`}, corresponding to Matlab operators $\{+, -, *, .*, ./\}$.

This is extremely useful because it allows an existing Matlab code executing an algorithmic application to effectively be re-used in order to perform AA-based floating-point error analysis for the same application.

Estimation of rounding error bound and range bound

To obtain the rounding error bound `ERR_BOUND` of a computational expression (cf. Section 3.5), the method `ERR_BOUND = ebound(Z_AA,prob)` is called, where `Z_AA` is the input `AAFloat` matrix, whose elements represents the affine forms of computed expressions. The optional parameter `prob` is used to define the confidence interval of the hard or probabilistic bound used. The `ebound` is an element-wise method that estimates the rounding errors for every element of affine forms, returning a `double` matrix `ERR_BOUND` (same size as `Z_AA`) of the rounding error bounds for all computed expressions in the floating-point matrix `Z`.

Similarly, the method `RANGE_BOUND = rbound(Z_AA,prob)` is used to estimate the upper bound for the range of each element in the affine form `Z_AA`, in which the optional parameter `prob` is used to define the confidence interval of the hard or probabilistic bound used. The `rbound` is an element-wise method which returns a `double` matrix `RANGE_BOUND` (same size as `Z_AA`) of the range bounds for all computed expressions in the floating-point matrix `Z`.

4.2.3 Special Affine Forms and Handling Special Cases

In this section, we discuss special affine forms for representing special intervals and how to handle special cases possibly occurring in the evaluation of AA expressions using the `AAFloat` class. There are two issues being discussed here:

- Representations of special affine forms
- Handling exceptional cases: division by zero and square root of negative range

Special affine forms

In the IA model, there exist two special intervals representing the value of a quantity x: the empty interval $[\ ]$, meaning "no value", and the real set interval $\mathbb{R}$, meaning "any real value" [53]. The definition of these special intervals is based on set theory, making it convenient for the evaluation of special mathematical operations. The interval $\mathbb{R}$ can be a result of a reciprocal or a division by an interval containing very small values, e.g., an interval has values very close to or equal zero. The empty interval $[\ ]$ can result from the evaluation of an interval having its lower bound larger than its upper bound, or from the evaluation of the square root of a negative interval.

An affine form implies a range for the quantity it represents. We denote the range represented by $\hat{x}$ with $[\hat{x}]$. If $\hat{x}$ is an affine form for the quantity x, then the value of x is guaranteed to be in the range of $\hat{x}$, i.e., $x \in [\hat{x}]$. There exist conversions between an affine form and the interval/range $[\hat{x}]$ implied by this affine form, for example, $[\hat{x}] = [x_0 - B(\hat{x}),\ x_0 + B(\hat{x})]$ is the range for the quantity x, in which the central value x_0 of the affine form $\hat{x}$ and the (hard or probabilistic) bounding operator $B(\cdot)$ are used to estimate the range of x.

Similar to IA, special affine forms for representing special intervals - the empty interval $[\ ]$ and the real set interval $\mathbb{R}$ - need to be defined in AA. More specifically, these special intervals need to be described explicitly in the `AAFloat` class.

In the `AAFloat` tool, floating-point arithmetic is used for representing affine expressions which imply respective affine intervals characterized by their lower and upper bounds. The IEEE-754 standard defines the special values: $+\infty$, $-\infty$ and Not-a-Number (NaN). Refer to [49] for the binary encoding of $+\infty$, $-\infty$ and NaNs. In the real set, the operations that can give rise to an NaN value may include: $\infty - \infty$, ∞/∞, $0 \cdot \infty$, and taking square root of a negative number. The infinity value can be the result of a division by zero. In Matlab, the floating-point arithmetic complies with the IEEE-754 standard, thus the special values for floating-point arithmetic are handled properly. This implies that the special values can possibly appear in the affine expression represented by the `AAFloat` object in Matlab. As the `AAFloat` class is based on floating-point arithmetic in Matlab, the representations of special affine forms (i.e., $\mathbb{R}$ and $[\ ]$) in the `AAFloat` class need to be established by convention.

In the `AAFloat` class, we use the NaN and ∞ values to represent the empty affine form $[\ ]$ and the real set affine form $\mathbb{R}$, respectively. Specifically, if the affine form $\hat{x}$ is an empty affine form, i.e., the interval implied by $\hat{x}$ is an empty range $[\hat{x}] = [\ ]$, then the range and error components of $\hat{x}$ are assigned to NaNs. Similarly, if the interval implied by $\hat{x}$ is the real set $[\hat{x}] = \mathbb{R}$, then the range and error components of $\hat{x}$ are assigned to ∞. Two examples for the representation of the empty interval and the real set $\mathbb{R}$ in the `AAFloat` class are shown below.

```
>> x_empty_interval
x_empty_interval =
AAFloat matrix of size [1 x 1]
(1,1)
    r: NaN
    e: NaN
    p: 0

>> x_R_interval
x_R_interval =
AAFloat matrix of size [1 x 1]
(1,1)
    r: Inf
    e: Inf
    p: 0
```

In fact, in the implementation of the `AAFloat` class, if any of the terms in the range or error component of an `AAFloat` object is a special value, then the corresponding affine expression will represent a special affine form, i.e., [ ] or $\mathbb{R}$.

Next, we present cases where special intervals arise in the affine expression in the `AAFloat` class. These cases can be divided into two usage scenarios: (a) when the arithmetic operations are performed, and (b) when the bounding operators are applied on the affine expressions containing the special interval. For handling these cases, we follow the algorithm and convention suggested by Stolfi and Figueiredo in [53, Chapter 3]. We describe how methods in the `AAFloat` class behave if any input interval is a special affine form.

For unary operations $z = f_1(x)$, i.e., the negation, inverse/reciprocal and square root operations, when the input $\hat{x}$ is a special affine form, i.e., $[\hat{x}] = [\ ]$ or $[\hat{x}] = \mathbb{R}$, then the output $\hat{z}$ is assigned to equal the input: $\hat{z} = \hat{x}$.

For binary operations $z = f_2(x, y)$, the `AAFloat` class will check whether any of the input operands is an empty interval first, then it will check whether any of the input operands is a real set interval. If $[\hat{x}] = [\ ]$ or $[\hat{y}] = [\ ]$, then the output is assigned to an empty affine form, i.e., $[\hat{z}] = [\ ]$. Otherwise, if $[\hat{x}] = \mathbb{R}$ or $[\hat{y}] = \mathbb{R}$, then the output is assigned to a real set affine form, i.e., $[\hat{z}] = \mathbb{R}$.

The behavior of the FMA operation when any of the three input operands happens to be a special affine form can also be inferred in a similar way.

When the bounding operator - a hard bounding or a probabilistic bounding - is applied to the affine expression containing the special interval, the resulting bound is an NaN or ∞ if the affine form contains NaN or ∞ values, respectively (i.e., corresponding to the empty interval [ ] or the real set interval).

Note that the special affine form $\mathbb{R}$ does not record any dependency information, meaning that if $[\hat{x}] = [\hat{y}] = \mathbb{R}$ then we cannot infer any constraint or relationship between the quantities x and y. Refer to [53] for a detailed discussion and algorithms for handling special affine forms.

Handling special cases

In interval arithmetic, there exist different design options for handling special cases, e.g., the division by zero where the input interval contains zero and the square root of an interval containing a negative range. In a simple interval model, an exception flag will be raised when these special cases happen. The wraparound model and loose evaluation [63] can also be chosen to handle the division by zero and the square root of a negative interval. Recently, the containment set theory has been suggested as a mathematical foundation for dealing with division by zero and special values, like $\pm\infty$, in interval arithmetic. See [63] and references therein for an extensive discussion. We do not use the containment set theory in this thesis.

In affine arithmetic, the design choices for handling division by zero and special values are more complex than the ones in interval arithmetic due to the fact that each affine form is an expression of multiple noise terms. The simplest design option for the division by zero and the square root of negative ranges is to raise an exception flag and then stop the current AA evaluation if these cases happen. For most cases, the simple design choice of raising an exception flag does not, however, provide any useful information to users. Our goal is to provide as much information as possible and offer an exception-free execution (EFE) [63] of affine expressions to the users of the `AAFloat` class. Therefore, we choose a more complicated yet more informative implementation for the inverse (division) and square root operations as follows.

For dealing with the inverse of AA forms containing zero, i.e., an input interval $[a, b]$ with $a \leq 0 \leq b$, the `AAFloat` class allows users to specify a reasonably true range $[c, d]$ for the input interval, with $a \leq c \leq d \leq b$, such that the range $[c, d]$ does not contain zero and is the largest range possible. The inverse of the original interval $[a, b]$ is, therefore, estimated via its representative interval $[c, d]$. As mentioned earlier in Chapter 3, the inverse operation for the input interval $[c, d]$ is estimated using the min-range approximation. As the interval $[c, d]$ does not contain zero while the original interval $[a, b]$ does, the inverse operation is, in fact, performed in one side of the original interval, the other side is neglected. This is just one design choice for handling division by zero situations in affine arithmetic in the `AAFloat` class. For range estimation, this design choice takes the larger range and sacrifices the smaller range for having a simple implementation for the resulting affine expression of the inverse. For rounding error estimation, this design choice is reasonable because the rounding error of a floating-point number depends on the magnitude of the number itself (see equs. 3.1-3.2).

We discuss how to determine the alternative interval $[c, d]$. The interval $[c, d]$ can be specified by users as the input argument `userrange` to the subroutine `div`, provided that users are able to determine the realistic range in practical applications. If the user does not provide the interval $[c, d]$, the `AAFloat` class will use the default range. The default value for $[c, d]$ is determined depending on the original interval $[a, b]$, which contains the zero value, and the working precision p. The larger range in magnitude is chosen. If $|a| > |b|$ then $[c, d] = [a, -2u]$, otherwise $[c, d] = [+2u, b]$, where $u = 2^{-p}$ is the unit roundoff.

The same idea is applied to handle the square root of a negative interval. If the input interval $[a, b]$ contains zero, i.e., $a \leq 0 \leq b$, then only the positive part of the input interval is evaluated. Similarly, the subroutine `sqrt` allows the user to specify a reasonably true range $[c, d]$ via the input argument `userrange`. If the user does not specify any range, the default values for $[c, d]$ are chosen. In theory, the lower bound of this range should be zero, as the square root of zero is defined, i.e., $\sqrt{0} = 0$. However, as the evaluation of the error component for the square root involves taking an inverse of the range component (see subsection 3.3.2), we choose to have $[c, d] = [+2u, b]$ for the square root operation in the `AAFloat` class.

The following simple examples, executed in the Matlab command window, compute the square root of an affine form $\hat{x}$ in the range $[-2, 5]$ at the single-precision ($p = 24$) format. The hard bounding operator is used. In the first operation computing `z1`, we do not specify the user's range, thus the default range corresponding to the working precision and the original interval is used. In the second operation computing `z2`, we specify the desired range of $[0.1, 5]$ for the square root operation. During the AA evaluation, the `AAFloat` class prints warning messages whenever the input interval to the square root operation contains a zero.

```
>> AAFloat_INIT
>> x = AAFloat(-2,5)
x =
AAFloat matrix of size [1 x 1]
(1,1)
    r: [1.500000000000000 3.500000000000000]
    e: 2.980232238769531e-07
    p: 24
>>
>> z1 = sqrt(x,1)
Warning: The input range [-2.0, 5.0] of the square root function contains NEGATIVE range.
The user does not specify any range for the input interval of the square root operation.
AAFloat is using the default range [1.192093e-07, 5.000000e+00] to compute the square root.
z1 =
AAFloat matrix of size [1 x 1]
(1,1)
    r: [1.397585670962136 1.117861355258394 0 0.279379048720740]
    e: [4.315837287514820e-04 0 1.665846154058737e-07]
    p: 24
>>
>> z2 = sqrt(x,1,[0.1, 5])
Warning: The input range [-2.0, 5.0] of the square root function contains NEGATIVE range.
z2 =
AAFloat matrix of size [1 x 1]
(1,1)
    r: [1.456661162929095 0.959920105741476 0 0 0 0.180513291170782]
    e: [4.712160915387251e-07 0 0 0 1.547988986874433e-07]
    p: 24
```

Demonstrative examples showing the practical use of the `AAFloat` tool for handling the division by zero (or inverse operation) in a real-world implementation of the Levinson-Durbin algorithm for linear prediction will be presented later in Section 4.4.

4.3 Error Verification via Simulations

The second task conducted in our framework is the simulation-based error verification. This task corresponds to the right path in Figure 4.1. The purpose of this task is to determine the maximum rounding error observed in the finite-precision implementation of the algorithm at hand at working precision p. Obviously, the error verification task is basically the custom-precision floating-point computation in Matlab.

The standard floating-point number formats supported by Matlab are double-precision and single-precision. So far, running bit-true floating-point computations using the standard Matlab library is still not possible. The challenging issue is, therefore, to efficiently perform custom-precision floating-point arithmetic and to integrate it in Matlab.

Implementation of MPFR-based custom-precision floating-point arithmetic in Matlab. For executing custom-precision floating-point operations in Matlab, we integrate arbitrary-precision arithmetic via the GNU MPFR Library [41] version 3.0.0 into our framework. MPFR C functions performing custom-precision operations are compiled into MEX files (Matlab EXecutable), which are then called within Matlab in the same way as Matlab files or built-in functions.

For current implementation of our framework, `mpfr_add`, `mpfr_mul`, `mpfr_div`, and `mpfr_fma` functions in the MPFR library were compiled to corresponding MEX functions in Matlab, allowing for custom-precision floating-point addition, multiplication, division and fused multiply-accumulate operations, respectively. These implemented MEX functions are sufficient for performing most of signal processing algorithms. Other custom-precision operations in the MPFR library, like floating-point square root, can be integrated into our framework in future implementations.

Determination of maximum simulation rounding error. The basic idea for determining the maximum rounding error is to run Monte Carlo simulations with a sufficiently large number of trials and determine the maximum error. Using more trials achieves more accurate rounding error estimates but also requires more simulation time. In our framework, 10^5 to 10^6 test samples (i.e., vectors or matrices) are randomly generated and executed for one test case of the algorithm.

We consider one *test case* of the algorithm corresponds to one chosen set of parameters and one working precision p. For each *test sample* within a test case, the rounding error is computed as the difference between the finite-precision floating-point result at precision p and its accurate (or infinite precision) reference computed at a very high precision, for which a double precision format is used if the precision p is smaller than double-precision and an 80-bit precision format is used if the precision p is very close to double-precision. The maximum simulation error for one test case is then defined as the maximum difference found among the total set of rounding errors fo all test samples.

Choice of different data distribution. In our framework, different data distributions for simulations can be chosen quite easily by using/setting different random number generators in Matlab. For example, users can use the `rand` function for generating uniformly distributed random numbers in one test case, and easily switch to the `randn` function for creating a normal distribution in another test case without having to change the computation part of the original Matlab program. Another example is presented in Chapter 5, in which the synthetic data for the reflection coefficients following a U-shaped distribution are generated for the error verification of the Levinson-Durbin algorithm (see 5.3.3). The advantage of easily choosing data distributions enables users to freely investigate the rounding error characteristics corresponding to different data distributions with very little programming effort.

In this work, the noise symbols ϵ_i of the AA forms are assumed to follow a uniform distribution over $[-1, +1]$, resulting in a normal distribution for AA forms once the probabilistic rounding operator is applied (refer to Section 3.2.3). We wish to understand the relation between the distribution of noise symbols and the distribution of the resulting AA form so that the desired distribution of an AA form can be specified by choosing the proper distributions for the noise symbols. This will allow for an AA-based rounding error estimation taking into account a given data distribution. However, that topic goes beyond the scope of this thesis; interested readers should refer to the work of Fang [4].

4.4 Examples

We present here two examples for practical use cases of the `AAFloat` class. Our goals include: *(i)* to show how to use the `AAFloat` class, given an original Matlab code, for performing rounding error analysis of floating-point algorithms; *(ii)* to show how to use the MPFR-based .mex functions for performing custom-precision floating-point computations based on the original Matlab code; and *(iii)* to show how the `AAFloat` class handles division by zero in an algorithm as well as *(iv)* to make users aware of cases where a division by zero may happen in an algorithm.

The two examples investigate the rounding error bounds for two floating-point algorithms: a sequential dot-product and the iterative Levinson-Durbin algorithm. As we will dedicate the entire Chapter 5 to floating-point error analysis and bit width allocation for the dot-product and the Levinson-Durbin algorithm, the demonstrative examples presented in this chapter only aim at showing the capability and usability of the `AAFloat` tool. An extensive discussion on reliability and accuracy of the AA-based error model for floating-point algorithms will be presented in Chapter 5.

4.4.1 AAFloat in Error Analysis of Sequential Dot-Products

In this example, we choose to estimate the rounding error of a sequential floating-point dot-product. The dot-product of two column vectors `x` and `y` of length `n` is a scalar `s` computed by multiplications and additions. The code for performing a sequential implementation of a floating-point dot-product in the single-precision or double-precision number formats (depending on the data type of input vectors) in Matlab is shown in Listing 4.1.

```
% ------------------------------------------------
% CODE 1: Sequential dot-product: Original code
% ------------------------------------------------
s = x(1) * y(1);
for i=2:n
    s = s + x(i) * y(i);
end;
```

Listing 4.1 Original Matlab Code

We perform error analysis with a chosen precision $p = 20$ (i.e., mantissa bit width). The length `n` of the input vectors is varied from 10 to 50 with a step size of 10. We assume that all elements `x(i)` and `y(i)` of the input vectors `x` and `y`, respectively, are in the range `[-1, 1]`.

Based on the original code shown above, two Matlab codes for rounding error estimation with the `AAFloat` class and for rounding error verification via extensive simulations with the MPFR-based Matlab .mex functions can be generated as follows.

Matlab Code for Rounding Error Estimation with the AAFloat Class. In this example, we apply the probabilistic bounding operator for the evaluation of AA forms, thus the optional parameter `prob` has its default value allowing for the use of standard Matlab operators (i.e., operator overloading). The Matlab code for the AA-based rounding error estimation with `AAFloat` is shown in Listing 4.2. The vector length `n` is set to 10 but can easily be changed to calculate other dot-products. The pair of functions `tic` and `toc` are for measuring the execution time of the code.

```
 1 % ---------------------------------------------------------------------
 2 % CODE 2: Sequential dot-product: Error bound estimation with AAFloat
 3 % ---------------------------------------------------------------------
 4 n = 10;                                  % vector length
 5 p = 20;                                  % precision
 6 a = 1; b = 1;                            % range
 7 AAFloat_INIT;
 8 x = AAFloat(-a*ones(n,1), a*ones(n,1), p);
 9 y = AAFloat(-b*ones(n,1), b*ones(n,1), p);
10 tic;
11 s = x(1) * y(1);                         % reuse the original code
12 for i=2:n                                % reuse the original code
13       s = s + x(i) * y(i);               % reuse the original code
14 end;                                     % reuse the original code
15 t = toc;
16 E_BOUND_s = ebound(s);
17 fprintf('n = %3d, p=%2d, E_BOUND_s = %e, t = %f (seconds)\n', n, p, E_BOUND_s, t);
```

Listing 4.2 Matlab Code for Rounding Error Estimation with the `AAFloat`

Taking a closer look into the code from line 11 to line 14 shows that the Matlab code using the `AAFloat` library reuses best the original code for the floating-point sequential dot-product algorithm as shown before.

Matlab Code for Rounding Error Verification via Simulations. The code for error verification via simulations is shown in Listing 4.3. In this code, the settings for the vector length, precision, numerical range and execution time measurement are identical to the Matlab code for error bound estimation with the `AAFloat` class.

For error verification, we run 10^6 test samples, specified by the variable `num_of_sample`. At each iteration corresponding to one testing sample, two input vectors are randomly generated by `rand`, the desired custom-precision value `s` at precision p and its reference value `s_ref` at double-precision are computed, and the rounding error is evaluated and stored. To perform custom-precision floating-point operations, two MPFR-based .mex functions, `mul_mpfr` and `add_mpfr`, are used for multiplication and addition, respectively, with the current working precision of each input operand specified when those functions are called.

The Matlab code for error verification (Listing 4.3) is more complex than the Matlab code for error bound estimation (Listing 4.2). The former can be executed with more memory, computations and conditional branches, and obviously it could take more time to run. Apart from that, it seems that the original Matlab code for sequential dot-product implementation is not well reused to generate the Matlab code for error verification in this example.

Table 4.3 Comparison between the `AAFloat` class and the simulation (with 10^6 samples) for the sequential dot-product

Vector length n	10	20	30	40	50
Execution time:					
- simulations [second]	39.38	47.33	54.55	67.02	74.92
- `AAFloat` [second]	0.03	0.06	0.09	0.12	0.15
- Speed-up	1.3e+3	7.9e+2	6.1e+2	5.6e+2	5.0e+2
Maximum number of error terms in the final AA form	70	140	210	280	350
OER of AA-based err. bound	2.11	1.85	1.93	1.79	1.57

```
% ---------------------------------------------------------------------------
% CODE 3: Sequential dot-product: Error verification by simulations
% ---------------------------------------------------------------------------
n = 10;                          % vector length
p = 20;                          % precision
a = 1; b = 1;                    % range
num_of_sample = 1e+6;            % number of trials
e = zeros(num_of_sample,1);
tic;
for k=1:num_of_sample
    x = -a + 2*a.*rand(n,1);
    y = -b + 2*b.*rand(n,1);
    s_ref = x' * y;
    s = mul_mpfr (x(1), p, y(1), p);
    for i=2:n
        s = add_mpfr (s, p, mul_mpfr(x(i), p, y(i), p), p);
    end;
    e(k) = s_ref - s;
end
t = toc;
e_max = max(abs(e));
fprintf('n = %3d, p=%2d, e_max=%e, t = %f (seconds)\n', n, p, e_max, t);
```

Listing 4.3 Matlab Code for Rounding Error Verification via Simulations

Performance Comparison. We compare the AA-based error bound estimation via the `AAFloat` class and the error verification via extensive simulations in terms of the execution times and accuracies of the rounding errors estimated.

To evaluate the performance, the speed-up in execution time of the Matlab code for AA-based error bound estimation with the `AAFloat` class compared to the Matlab code for extensive simulations is reported. All the Matlab codes, the code using `AAFloat` class and the code running extensive simulations, are run on an AMD Athlon 3800 Dual Core desktop CPU at a clock frequency of 2.0 GHz. The number of error terms in the final AA forms are also reported. To evaluate the accuracy of the AA-based rounding error bound (i.e., a probabilistic bounding operator with the confidence interval of 3σ is used in this example), the over estimation ratio *OER* is calculated (see Equation (4.1)).

Table 4.3 presents the accuracy and performance comparison versus the vector length n. With respect to the execution time, the AA-based error bound estimation with the `AAFloat` tool can gain a speed-up from two to three orders of magnitudes compared to simulations. With respect to the accuracy, the rounding error bounds estimated by `AAFloat` are about 1.57 to 2.11 times larger than the maximum errors evaluated by running simulations. In terms of bit width allocation, the reported *OER*s correspond to an over estimate of about 1 mantissa bit for the hardware implementation of a sequential floating-point dot-product. Note that there is a trade-off between the number of test samples and the execution time for simulations. We have tried with different values of test samples. Our experiments show that 10^6 samples are sufficient to find the maximum value of the rounding errors of the floating-point dot-products.

The number of error terms in the final computed AA form accounts for both the range component and the rounding error component and includes the central value in the range component. Given the vector length n, the generation of vectors `x` and `y` needs $(2n + 1)$ and $2n$ terms for the range and error components, respectively. For the computation, the sequential dot-product implementation requires n multiplications and $(n - 1)$ additions. Each multiplication generates one new noise term in the range component and one new noise term in the rounding error component, while each addition only generates one new noise term in the error component. This results in a total number of n new error terms for the range component and $(2n - 1)$ new noise terms for the error component for the whole computation of the sequential dot-product. Eventually, the final computed AA form of the dot-product consists of $(3n+1)$ and $(4n-1)$ error terms in the range and rounding error components, respectively. In other words, the final AA form has $7n$ error terms in total (including the central value in the range component). Obviously, the number for terms scales linearly with the vector length. For this demonstrative example, the maximum number of error terms is 350 at vector length $n = 50$. there are 350 error terms in the final AA form.

Note that an in-depth study of the rounding errors of different floating-point dot-product implementation variants versus a wider range of precision and vector length is presented in Chapter 5. In that study, the maximum number of error terms in the final AA form is about 7000.

4.4.2 AAFloat in Error Analysis of Levinson-Durbin Algorithm

The Levinson-Durbin algorithm [7, 8] is an efficient algorithm for solving Yule-Walker equations and it is often used for linear prediction in speech processing applications. Given a system order n of the Yule-Walker equations, the Levinson-Durbin algorithm receives an autocorrelation coefficient vector $\mathbf{r0} = [1, r_1, r_2, \dots r_n]^T$ as its input and computes the filter coefficients a_i, reflection coefficients k_i and the short-term prediction error E_i $(i = 1 \dots n)$ in an iterative manner. A detailed description of the Levinson-Durbin iterations is presented later in subsection 5.3.2. The autocorrelation coefficients satisfy $r_0 = 1$ and $-1 \leq r_i \leq 1$, $i = 1 \dots n$. The prediction error E_i is bounded as

$$0 \leq E_i \leq 1, \quad 0 \leq E_i \leq E_{i-1} \leq E_0 = r_0 = 1$$

and the reflection coefficients of a minimum-phase predictor system are bounded as follows [7, 30]

$$-1 \leq k_i \leq 1, \quad i = 1 \dots n.$$

At the i-th iteration, the evaluation of the reflection coefficient k_i involves taking the inverse of the prediction error E_{i-1} computed at the previous iteration, possibly leading to a division-by-zero exception if E_{i-1} comes close to zero.

In AA-based error analysis, the range of E_{i-1} continuously expands over iterations and/or with the magnitude of the autocorrelation coefficient r_i. Therefore, it is very likely that the AA-based range for the prediction error may contain zero and the division by zero can happen. See subsection 5.3.5 for more discussion on the implications of the prediction error as well as on the accuracy of the AA-based error model for the Levinson-Durbin algorithm.

Our concern here is to know how the `AAFloat` class handles the division by zero case if the prediction error E_{i-1} does contain the zero value in its range when performing AA-based error analysis of the Levinson-Durbin iterations. We would also like to learn how the coefficients will come out and what the user should do if a division-by-zero happens.

In the following, we investigate three use cases of the `AAFloat` tool for AA-based error analysis of the Levinson-Durbin algorithm:

- *Case 1: No division by zero.* The range of the prediction error does not contain zero. The AA-based evaluation of the algorithm is executed without any warning or error message.
- *Case 2: Default setting.* The range of the prediction error contains zero. The division by zero happens and the `AAFloat` tool uses the default setting for the smaller bound in magnitude.
- *Case 3: User's setting.* The range of the prediction error contains zero. The division by zero happens and the user specifies the smaller bound in magnitude to be used by the `AAFloat` tool.

For simplicity, we use the hard bounding operator for all the three use cases presented.

Case 1: No division by zero

We form a Yule-Walker equation of order $n = 10$ using a restricted range for the input autocorrelation coefficients r_i. The working precision is chosen as $p = 24$ (i.e., single-precision). In this example, each autocorrelation coefficient r_i is assumed to be uniformly distributed over the range $[0.17,\ 0.23]$, such that their AA forms are shown as

$$\hat{r}_0 = 1, \quad \hat{r}_i = 0.2 + 0.03\epsilon_i, \quad \epsilon_i \in [-1, +1], \quad i = 1 \dots n.$$

We observed that the `AAFloat` class estimates reasonably the range and error bounds for all the quantities, i.e., E_i, k_i, and a_i, in the Levinson-Durbin iterations in comparison with the experimental results by simulations as well as the theoretical range bound derived in the literature. See Appendix E.1 (from iteration 1 to 10) for a full report of the `AAFloat` class for the use case 1.

Case 2: Default setting

Now we increase the system order up to $n = 15$ but keep the same range $[0.17,\ 0.23]$ for the autocorrelation coefficients r_i. The full report of the `AAFloat` class for the use case 2 is listed in Appendix E.1. We observe that the AA-based range for the prediction error E_i increases with the iteration, reported by the `AAFloat` tool and shown in Table 4.4.

At the end of iteration 12, the range for the prediction error E_{12} is $[-0.55,\ 2.24]$, i.e., containing zero, generating a division by zero in the inverse operation at iteration 13. The `AAFloat` tool reports as follows:

```
---------- Iteration n = 13
Levinson-Durbin algorithm for special case [0.17, 0.23]
```

Table 4.4 Range of prediction error E_i of the Levinson-Durbin iterations reported by the `AAFloat` tool for the use case 2

Iteration i	Range of E_i	Remark
9	$[0.72,\ 1.00]$	
10	$[0.63,\ 1.08]$	
11	$[0.42,\ 1.29]$	
12	$[-0.55,\ 2.24]$	Division by zero at iteration 13!
13	$[-2.8 \times 10^7,\ 2.8 \times 10^7]$	Division by zero at iteration 14!

```
AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 1.149381e-05, range = [-0.546389, 2.243222]

Warning: The input range [-0.546389, 2.243222] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1072
  In AA_bound_levinson_wAAFloat_simple_example_increase_order at 88
The user does not specify any range for the input interval of the inverse operation.
AAFloat is using the default range [1.192093e-07, 2.243222e+00] to compute the inverse.
...
i=13, error bound of k(13) = 1.191544e+09, range = [-12358221.218054, 11952076.988843]
...
error bound of E_{m} = 3.072350e+09, range = [-27933964.201552, 27914301.953442]
```

As we do not specify any lower bound (in absolute value) for the range of E_{12}, the `AAFloat` class uses its default value of $2u = 1.192093 \times 10^{-7}$, where $u = 2^{-p}$ is the unit roundoff corresponding to the current working precision $p = 24$. This is one of the very useful features of the `AAFloat` tool in handling division by zero, that the tool both warns the user whenever a division by zero happens and it automatically continues the execution of the AA evaluation process.

When a division by zero happens in the algorithm and the default bound is then used by the `AAFloat` tool, we would like to know how far the AA evaluation process can go and whether the resulting bound is reasonable compared to its expected range and/or to experimental results by simulations. In the report for the use case 2 above, the resulting range and error bounds for the reflection coefficient k_{13} at iteration 13 are in the orders of 10^7 and 10^9, respectively, which are extremely far away from the possibly meaningful and reasonable ranges for a reflection coefficient of the Levinson-Durbin algorithm. In this situation, continuing the error evaluation will give non-meaningful results.

Case 3: User's setting

In the last use case of the `AAFloat` tool for the Levinson-Durbin algorithm, we specify the user's range for the prediction error E_i which is used by the `AAFloat` class when the division by zero occurs. The system order, the working precision and the input range for the autocorrelation coefficients are the same as in the second use case, i.e., $n = 15$, $p = 24$ and $r_i \in [0.17,\ 0.23]$. By specifying the user's range, we hope that better range and error bounds for the quantities in the Levinson-Durbin algorithm could be obtainable.

The important question is to identify the range for the prediction error to be used as the input to

Table 4.5 Range of prediction error E_i of the Levinson-Durbin iterations reported by the `AAFloat` tool for the use case 3

Iteration i	Range of E_i	Remark
11	$[0.42,\ 1.29]$	
12	$[-0.55,\ 2.24]$	Division by zero at iteration 13!
13	$[-3.3 \times 10^4,\ 3.3 \times 10^4]$	Division by zero at iteration 14!
14	$[-1.3 \times 10^{13},\ 1.3 \times 10^{13}]$	Division by zero at iteration 15!

Table 4.6 Comparison of the range and error bounds for the reflection coefficient k_{13} between the use case 2 and the use case 3

	Range bound of k_{13}	Error bound of k_{13}
Use case 2: Default setting	$[-1.2 \times 10^7,\ 1.2 \times 10^7]$	1.2×10^9
Use case 3: User's setting	$[-1.5 \times 10^4,\ 1.4 \times 10^4]$	1.7×10^3

the inverse operation in the Levinson-Durbin algorithm. The smallest value of the prediction error, reported by running simulations of 10^6 data frames of speech signal and synthetic data, is in the order of 10^{-4}. We therefore choose the lower bound for the prediction error E_i as 10^{-4}. At each iteration, we use the current upper bound for the range of the prediction error E_i as the user-specified upper bound for E_i.

The full report of the `AAFloat` class for this use case is listed in Appendix E.3. Table 4.5 reports the range of the prediction error E_i over some iterations. By using the user-specified range, the AA-based error evaluation process using the `AAFloat` tool for the Levinson-Durbin iterations can execute up to iteration 14. The evaluation process fails and stops at iteration 15. With respect to the resulting bounds for the range and the rounding error, the use case using a user-specified range for the prediction error (use case 3) can provide less overestimated bounds compared to the use case with the default range of the tool (use case 2), reported in Table 4.6. At iteration 13, the range and error bounds for the reflection coefficient k_{13} are approximately 10^4 and 10^3, respectively, compared to the respective resulting ranges of 10^7 and 10^9 reported in the second use case (see Appendix E.3 and Appendix E.1 for the detailed reports of the use cases 3 and 2). Note that a rounding error bound of 10^3 for the reflection coefficient at a single-precision number format is still very pessimistic compared to the realistic error in practical applications. However, discussions on the accuracy of the AA-based error model for complex algorithms like the Levinson-Durbin iteration are not the main focus of this chapter and, therefore, will be reserved for Chapter 5.

The AA-based error analysis for the Levinson-Durbin algorithm presented in this section only shows typical use cases of the `AAFloat` tool in handling special cases, like the division by zero, which could possibly happen in any floating-point algorithm. In the `AAFloat` tool, we try to provide as much information as possible to the users whenever an exceptional case occurs. A more important issue the users of the `AAFloat` tool should be well aware of is understanding and interpreting the meanings of the resulting range and error bounds, reported by the tool, in the floating-point algorithm at hand. If a special case happens, e.g., division by zero, the `AAFloat` tool can help provide some guidelines for users. It is, however, the users' decision to use the default range of the tool, or to continue the AA-based evaluation using a new specified (and more reasonable) range, or to stop the current evaluation process and modify the algorithm at hand, based on the information provided by the tool.

4.5 Conclusions

This chapter presented our implementation of the first Matlab-based tool, the `AAFloat` class, for partially automated rounding error and numerical range evaluation of floating-point algorithms using affine arithmetic error modeling. Given an original Matlab code performing any floating-point algorithm, a user needs to convert floating-point variables into `AAFloat` objects (by using the `AAFloat` class constructor method) before performing the error evaluation of the same algorithm following the AA-based error model. The computational expressions in the original Matlab code can be reused. By supporting basic vector and matrix computations in Matlab's notation, the framework enables users to reuse their own existing Matlab codes to effectively perform the rounding error analysis task.

Besides, the framework incorporates custom-precision floating arithmetic via the GNU MPFR library, allowing for the bit-true custom-precision implementation of floating-point arithmetic in Matlab.

Moreover, in the `AAFloat` tool we implement a new technique for handling exceptional cases. With this improvement, the `AAFloat` tool gives useful information and allows users to specify reasonable ranges in handling the division by zero and square root of negative intervals.

With the benefits of performance speed-up, accurate error bound estimation and flexible handling of exceptional cases, the AA-based error analysis technique in combination with the `AAFloat` tool can be an alternative to simulation-based error analyses.

In the next chapter, we will present in detail how this Matlab-based framework is used to perform rounding error analysis and bit width allocation of the two important floating-point signal and speech processing applications: the dot-product and the Levinson-Durbin algorithm.

5 Applications

5.1 Introduction

As aforementioned in Chapter 3, existing rounding error analysis techniques exhibit some drawbacks with respect to the tightness of estimated bounds and estimation time. Conventional rounding error analysis techniques often provide very pessimistic rounding error bounds. The dynamic rounding error analysis techniques based on simulations can provide much tighter error bounds compared to their conventional counterpart, but require very long simulation times and are data dependent. Therefore, these above rounding error analysis techniques are often suboptimal for bit width allocation.

We ask if AA-based error analysis techniques are able to overcome these drawbacks, i.e., whether AA-based error analysis allows for deriving in less time error bounds comparable to those obtained by simulations, such that the AA-based error bound derived can be used for bit width allocation for hardware implementations. Second, we would like to show the efficiency of the `AAFloat` class and Matlab-based framework presented in Chapter 4 for rounding error estimation of floating-point algorithms. To answer the two questions above, this chapter will apply AA-based error analysis techniques for the estimation of floating-point rounding errors of two signal and speech processing applications.

In the first application, we try to estimate the rounding errors of different floating-point dot-product implementation variants versus precision. Our Matlab-based framework is used to evaluate the AA probabilistic error bounds via the `AAFloat` class, and to experimentally estimate the maximum rounding errors via simulations. The *error bounds* and *calculation time* of AA-based and simulation-based error analysis techniques are compared.

In the second application, we use the AA-based error model and our software tool to perform floating-point rounding error analysis for the iterative Levinson-Durbin algorithm.

5.2 Rounding Error Analysis of Floating-Point Dot-Products

5.2.1 Motivation

The dot-product is an important subroutine in signal processing applications, especially for the implementation of Finite Impulse Response (FIR) filters. In scientific computing, the dot-product is the basic functional block for matrix-vector and matrix-matrix multiplication. Rounding error analysis for floating-point dot-products has been studied by [5, 64] using conventional floating-point error

Table 5.1 Names of dot-product implementations considered

	Basic operations	Fused operation (FMA)
Sequential architecture	`SeqDot`	`SeqDotFMA`
Parallel architecture	`ParDot`	*not considered here*

models. Using AA-based error analysis techniques, related work in [4] estimated the rounding error bound for floating-point FIR filter implementations, which is however only *a special case of the general dot-product* implementation because one of the input operands is a constant vector representing the coefficients of the FIR filter.

Recently, Mücke *et al.* [13] investigated the performance model for custom-precision floating-point dot-products on FPGAs, showing that superlinear gains in peak performance and parallelism can be achieved by reduced mantissa bit width operands in a binary-tree dot-product architecture. However, an open research question is the identification of the optimal mantissa bit width.

This work uses the AA-based probabilistic bounding operator to estimate the rounding error bounds of different floating-point dot-product architectures. The validity of the estimated error bounds is demonstrated using extensive simulations. We derive the analytical models for rounding errors of floating-point dot-products over a wide range of parameters. We also compare the AA-based rounding error bound with the conventional error bound, and perform comparisons among different dot-product architectures.

5.2.2 Dot-Product

Given two column vectors $\mathbf{x}$, $\mathbf{y}$ of length n: $\mathbf{x} = [x_1, ..., x_n]^T$, $\mathbf{y} = [y_1, ..., y_n]^T$, the dot-product of $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$\mathbf{x}^T\mathbf{y} = \sum_{i=1}^{n} x_i \cdot y_i = x_1y_1 + ... + x_ny_n.$$

Different dot-product implementation variants result in different rounding errors [5]. In this work, we perform error analysis for two different floating-point dot-product architectures: a sequential dot-product and a parallel (having binary-tree adders) dot-product [13]. We investigate the rounding error with respect to the types of floating-point operations used: basic operations (i.e., multiplication and addition) and a fused operation FMA.

For the sequential dot-product architecture, two implementation variants are considered: a sequential dot-product using basic floating-point operations (`SeqDot`) and a sequential dot-product using FMA (`SeqDotFMA`). For the parallel dot-product architecture, we consider only the implementation variant using the basic floating-point operations (`ParDot`)[1].

In total, three dot-product implementation variants are investigated in this work. Table 5.1 presents the names of three dot-products and Figure 5.1 presents their block diagrams.

[1]We do not consider the parallel dot-product using FMA, in which the multiplications are performed by FMAs while the additions are performed by adders, because this implementation variant does not explore the FMAs for adding, therefore there is no chance to improve the final accuracy compared to using a `ParDot` implementation.

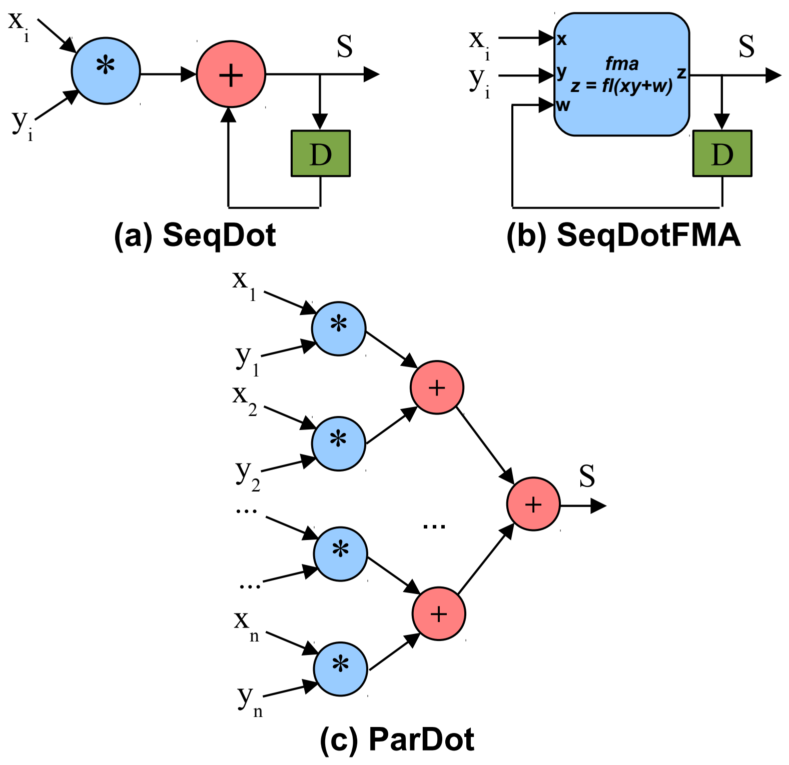


Figure 5.1 Dot-product implementation variants

5.2.3 Experimental Setup

Simulation setup

We choose symmetric numerical ranges for the inputs as $x_i \in [-a, +a] = [-128, +128]$, and $y_i \in [-b, +b] = [-128, +128]$. The precision p (mantissa bit width) of the floating-point operands varies in unit steps from 20 bits up to 53 bits (double-precision). The exponent bit width is fixed as 11 bits. The vector length n of the two input vectors ranges from 100 to 1000 with a step size of 100. The chosen confidence interval [4] for the AA probabilistic bounding operator is 3σ, corresponding to a probability of 99.73% that the rounding error will fall within the AA probabilistic bound. All the simulations are run on an AMD Athlon 3800 Dual Core desktop CPU at a clock frequency of 2.0 GHz.

Matlab codes for AA-based error estimation of dot-product variants using `AAFloat`

Using the `AAFloat` class, we implement two Matlab functions, `AASeqDot` and `AASeqDotFMA`, for AA-based rounding error estimation of sequential dot-products using basic operations and fused multiply-add operation, respectively. The Matlab codes for sequential dot-product implementations are quite simple and shown below, in which `a` and `b` define the symmetric numerical ranges of the input vectors, `n` is the vector length, `p` is the precision and `prob` defines the bounding operator.

```
function SBOUND = AASeqDot(a, b, n, p, prob)
    AAFloat_INIT;
    x = AAFloat(-a*ones(n,1), a*ones(n,1), p);
    y = AAFloat(-b*ones(n,1), b*ones(n,1), p);
    s = mmul(x', y, prob);
    SBOUND = ebound(s, prob);
end
```

```
function SBOUND = AASeqDotFMA(a, b, n, p, prob)
    AAFloat_INIT;
    x = AAFloat(-a*ones(n,1), a*ones(n,1), p);
    y = AAFloat(-b*ones(n,1), b*ones(n,1), p);
    s = 0;
    for k=1:n
        s = fma (x(k), y(k), s, prob);
    end
    SBOUND = ebound(s, prob);
end
```

For the parallel dot-product implementation having $M = \lceil log_2 n \rceil$ stages of binary-tree adders, the Matlab codes for rounding error estimation using the `AAFloat` class are a little more complicated because n multiplications are computed (in parallel) first then the summations are conducted in a pairwise manner within M adding stages. The Matlab function for the parallel dot-product using basic operations `AAParDot` is shown below.

```
function SBOUND = AAParDot(a, b, n, p, prob)
    AAFloat_INIT;
    x = AAFloat(-a*ones(n,1), a*ones(n,1), p);
    y = AAFloat(-b*ones(n,1), b*ones(n,1), p);
    s_prev = mul(x, y, prob);
    M  = ceil(log2(n));       % number of adding stages
    for m=1:M
        N = floor(length(s_prev)/2);
        s = add (s_prev(1:N), s_prev(N+1:2*N), prob);
        if mod( length(s_prev), 2)
            s = [s; s_prev(2*N+1)];
        end;
        s_prev = s;
    end;
    SBOUND = ebound(s, prob);
end
```

We use the *Over-Estimation-Ratio* (OER), defined by Equ. (4.1) in 4.1.3, to evaluate the tightness of the estimated AA error bounds for different dot-product variants.

5.2.4 Experimental Results

Rounding errors of sequential dot-products

Figure 5.2 presents the contour maps of the maximum rounding error (dashed lines) obtained by simulations and the AA probabilistic rounding error bound (solid lines) for the sequential dot-product

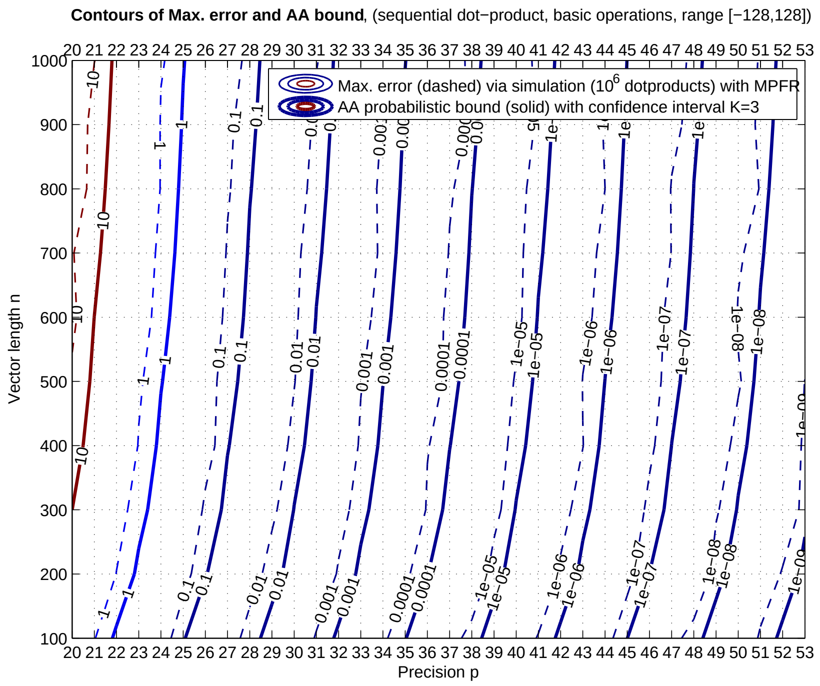


Figure 5.2 Contours of maximum rounding error and AA probabilistic bound for the sequential dot-product using basic operations (`SeqDot`)

using basic operations (`SeqDot`). Similarly, Figure 5.3 shows the contours of maximum rounding errors and respective AA probabilistic bounds of the sequential dot-product using the fused operation `SeqDotFMA`. For both sequential dot-products, without or with FMA, we observe that, regardless of the vector length or precision, AA probabilistic bounds are very close to realistic errors via simulations and the OER_{seq} of the AA probabilistic bounds for the sequential dot-products is always within the range: $1.2 \leq OER_{seq} \leq 2.2$.

A very interesting observation when comparing Figure 5.2 and Figure 5.3 is that the sequential dot-product using basic operations (multiply & add) and the sequential dot-product using fused operation (FMA) have the very same error patterns. For the same chosen vector length n and precision p, the two dot-product implementations produce almost the same rounding errors. This can be explained as follows. The error model of the fused operation (see Equ. 3.31) shows us that an FMA can only eliminate the rounding error due to multiplication, but cannot reduce the rounding error due to addition. During the adding process for computing dot-products, the accumulating sum grows up significantly and therefore the rounding errors due to additions are the dominant errors, leading to a final rounding error very similar to the final rounding error of the dot-product using basic operations.

Our conclusion is that, for the sequential dot-products, using an FMA cannot improve the overall numerical accuracy (compared to using basic floating-point operations).

Rounding errors of the parallel dot-product

For the parallel dot-product using basic operations, the contours of the rounding errors by simulations and the respective AA error bounds are presented in Figure 5.4 We observe that, regardless of the

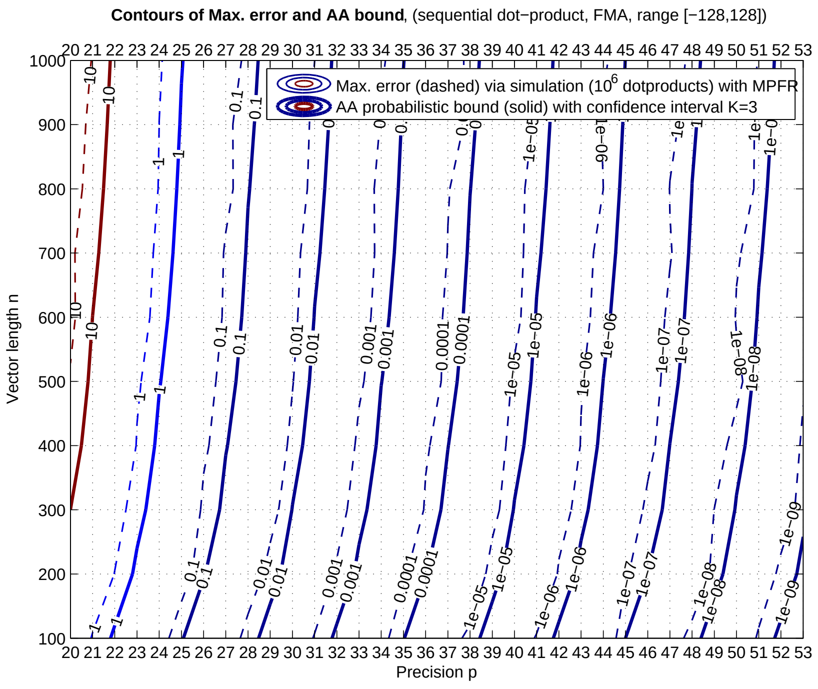


Figure 5.3 Contours of maximum rounding error and AA probabilistic bound for the sequential dot-product using FMA (`SeqDotFMA`)

vector length or precision, the OER_{par} of the AA probabilistic bound for the parallel dot-product is slightly larger compared to the OER_{seq} of the sequential dot-products, yet always within the range: $1.3 \leq OER_{par} \leq 4.1$.

Comparison between sequential and parallel dot-products

We compare the rounding errors of two different dot-product architectures: the sequential dot-product using basic operations (`SeqDot`, Figure 5.2) versus the parallel dot-product using basic operations (`ParDot`, Figure 5.4). It is not hard to see that the parallel dot-product offers more accurate final results as all error curves in Figure 5.4 move to the left, compared to the error curves of the sequential dot-product in Figure 5.2. Moreover, due to the binary-tree structure, the errors of parallel dot-product increase like $log_2 n$ (where n is the vector length) and, therefore, are less dependent on the vector length in comparison with sequential dot-products.

For bit width allocation, the AA probabilistic bounds for all the investigated dot-products are reliable and efficient estimates, corresponding to the over-estimation of mantissa bit widths of at most $log_2(2.2) \approx 1$ bit for sequential dot-product implementations, and $log_2(4.1) \approx 2$ bits for the parallel dot-product implementation, respectively.

Calculation time

Table 5.2 reports the time (in seconds) required for analyzing maximum errors via simulation of 10^6 samples, and for estimating AA probabilistic bounds of all dot-products implementations with the smallest vector length of $n = 100$ and at single-precision format ($p = 24$). Note that the calculation time does not depend on a specified precision. In terms of calculation time, the AA-based approach

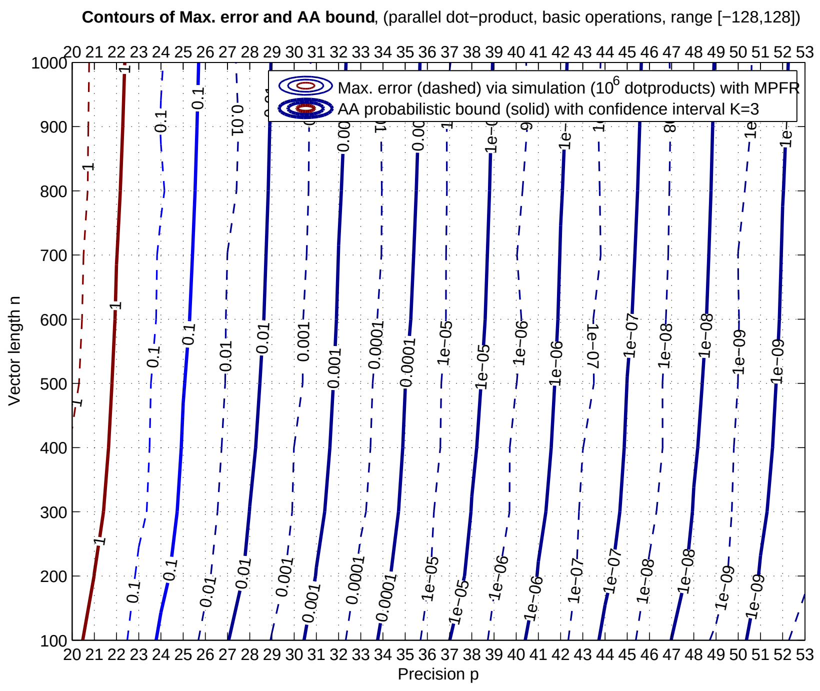


Figure 5.4 Contours of maximum rounding error and AA probabilistic bound for the parallel dot-product using basic operations (`ParDot`)

Table 5.2 Execution time of rounding error estimation ($p = 24, n = 100$)

	Simulation [s] (10^6 samples)	AA [s]	Speedup
`SeqDot`	230	0.32	7.2×10^2
`SeqDotFMA`	350	0.42	8.3×10^2
`ParDot`	3041	0.46	6.6×10^3

achieves a speedup from two to three orders of magnitude compared to the simulation-based approach. This reveals that the AA-based approach can be used as a computationally efficient and yet reliable alternative for the simulation-based approach.

5.2.5 Analytical Error Models of Floating-Point Dot-Products

In this section, going a step further in AA-based error analysis, we aim to derive some *analytical expressions* for the error bounds of floating-point dot-products as a function of numerical range, precision p and vector length n by using an AA probabilistic bounding operator. Our derivation of analytical error bound expressions for floating-point dot-products is based on the following assumptions:

- Each element of the two input vectors is uniformly randomly distributed in a symmetric range, i.e., $x_i \in [-a, +a], y_i \in [-b, +b], i = 1, .., n$. As a consequence, the AA form representing each input operand has a central value of zero.

- ❑ The relative rounding errors δ_i due to floating-point computations are independent random variables that follow a uniform distribution in the range $[-u, +u]$, from which an AA probabilistic bounding operator can be applied.

Note that the analytical expressions for floating-point dot-products derived in this section are basically the more compact and convenient forms of AA probabilistic error bounds that have been estimated by using the `AAFloat` tool in previous sections. Analytical expressions, however, allow for convenient comparison of the AA error analysis technique and the conventional error analysis technique, which will be presented later on.

As presented in Section 3.3, we represent each floating-point quantity by a numerical range component and a rounding error component. For example, the input operand x_i can be represented by an AA form $\hat{x}_i = \hat{x}_i^r + \hat{x}_i^e$. The magnitude of the rounding error component is closely related to the magnitude of the range component and can be determined by applying the bounding operator $B(\cdot)$. In the following analytical derivation, we use the AA probabilistic bounding operator.

AA form of input operands

The AA forms of input operands x_i, y_i are given by

$$\begin{aligned}\hat{x}_i &= \hat{x}_i^r + \hat{x}_i^e = (0 + a \cdot \epsilon_{xx_i}) + (a \cdot u \cdot \epsilon_{x_i}),\\ \hat{y}_i &= \hat{y}_i^r + \hat{y}_i^e = (0 + b \cdot \epsilon_{yy_i}) + (a \cdot u \cdot \epsilon_{y_i}),\end{aligned}$$

in which $u = 2^{-p}$ is the unit roundoff, p is the current working precision; ϵ_{xx_i}, ϵ_{yy_i} are the noise symbols associated the range components and ϵ_{x_i}, ϵ_{y_i} are the noise symbols associated with quantization errors of the floating-point numbers x_i, y_i. Note that the central values of the range components of x_i, y_i equal zero because the input operands are assumed to be uniformly distributed in symmetric ranges, i.e., $x_i \in [-a, +a], y_i \in [-b, +b], i = 1, .., n$.

AA form of product $P_i = x_i y_i$

The AA form of product $P_i = x_i y_i$ evaluated by a basic floating-point multiplication, as is the case for the `SeqDot` and `ParDot` implementations, is derived as

$$\begin{aligned}\hat{P}_i &= \hat{P}_i^r + \hat{P}_i^e\\ &= (0 + ab\epsilon_{pp_i}) + (abu\epsilon_{x_i} + abu\epsilon_{y_i} + B(\hat{P}_i^r)u\epsilon_{P_i})\\ &= (0 + ab\epsilon_{pp_i}) + (abu\epsilon_{x_i} + abu\epsilon_{y_i} + abu\epsilon_{P_i}),\end{aligned}$$

where ϵ_{pp_i} is the noise symbol associated with the range component and ϵ_{P_i} is the noise symbol associated with the rounding error component of the AA form $\hat{P}_i$. Note that for simplicity, we skipped the second-order rounding error terms for deriving the AA form $\hat{P}_i$.

Looking into the error component $\hat{P}_i^e$ of the product P_i we see that when multiplying two floating-point operands $P_i = x_i y_i$, the quantization error of each operand is multiplied with the other operand, making the rounding error of the product P_i grow larger. Hence, each product $P_i = x_i y_i$ introduces an error expression consisting of $abu\epsilon_{x_i}$ due to x_i, $abu\epsilon_{y_i}$ due to y_i, and $abu\epsilon_{P_i}$ due to the rounding of the floating-point multiplication itself, which can be determined by applying the bounding operator $B(\hat{P}_i^r)$.

Table 5.3 Expressions for analytical derivation of the AA bound for the sequential dot-product using multiplication & addition ($u = 2^{-p}$; $x_i \in [-a, +a], y_i \in [-b, +b]$)

Quantity	AA expression of rounding error component
x_i	$au\epsilon_{x_i}$
y_i	$bu\epsilon_{y_i}$
$P_i = x_i \cdot y_i$	$abu\epsilon_{x_i} + abu\epsilon_{y_i} + abu\epsilon_{P_i}$
$S_1 =$ $P_1 + P_2$	$abu\epsilon_{x_1} + abu\epsilon_{y_1} + abu\epsilon_{P_1}$ $+abu\epsilon_{x_2} + abu\epsilon_{y_2} + abu\epsilon_{P_2} + B(\hat{S}_1^r)u\epsilon_{S_1}$
$S_2 =$ $(P_1 + P_2)$ $+P_3$	$abu\epsilon_{x_1} + abu\epsilon_{y_1} + abu\epsilon_{P_1}$ $+abu\epsilon_{x_2} + abu\epsilon_{y_2} + abu\epsilon_{P_2} + B(\hat{S}_1^r)u\epsilon_{S_1}$ $+abu\epsilon_{x_3} + abu\epsilon_{y_3} + abu\epsilon_{P_3} + B(\hat{S}_2^r)u\epsilon_{S_2}$
$S_{n-1} =$ $x_1y_1 + x_2y_2$ $\cdots$ $+x_ny_n$	$abu\epsilon_{x_1} + abu\epsilon_{y_1} + abu\epsilon_{P_1}$ $+abu\epsilon_{x_2} + abu\epsilon_{y_2} + abu\epsilon_{P_2} + B(\hat{S}_1^r)u\epsilon_{S_1}$ $\cdots$ $+abu\epsilon_{x_n} + abu\epsilon_{y_n} + abu\epsilon_{P_n} + B(\hat{S}_{n-1}^r)u\epsilon_{S_{n-1}}$

If the product P_i is computed by using a fused operation FMA, as is the case for `SeqDotFMA`, the rounding errors due to multiplications are avoidable for most of the cases,[2] which leads to the following AA form for the product x_iy_i computed by a fused operation FMA

$$\begin{aligned}\hat{P}_{i,fma} &= \hat{P}_{i,fma}^r + \hat{P}_{i,fma}^e \\ &= (0 + ab\epsilon_{pp_i}) + (abu\epsilon_{x_i} + abu\epsilon_{y_i}).\end{aligned}$$

General AA expression for rounding error component of dot-product

The AA expression for the final rounding error of dot-products equals the sum of all rounding error components of products P_i, $i = 1, 2, \ldots n$. In the following, we will first detail the AA expression for the final rounding error of the sequential dot-product using basic operations `SeqDot`. We will then derive a general AA expression for the rounding error component of any floating-point dot-product implementations considered in this work.

Table 5.3 details AA expressions of the rounding error components of quantities used to compute the sequential dot-product using basic operations. For sequential implementation, $(n-1)$ additions are performed sequentially, i.e., one addition per time, and the rounding errors of products P_i are accumulated. More importantly, each addition for updating the sum introduces one new rounding error component $B(\hat{S}_i^r)u\epsilon_{S_i}$, $i = 1, 2 \cdots (n-1)$, whose magnitude depends on the magnitude of the current sum and is determined by taking the AA probabilistic bounding operator $B(\hat{S}_i^r)$ then multiplying with the unit roundoff u.

From Table 5.3 we can easily rewrite the AA expression for the final rounding error component of

[2] If we assume that, in the `SeqDotFMA` implementation, the input operand w to an FMA $z = \texttt{fma}(xy + w)$ is a non-zero number, then the multiplication can be computed without error.

Table 5.4 Rounding error components of four dot-product implementations

Implementation	$\hat{E}_x$	$\hat{E}_y$	$\hat{E}_{mult}$	$\hat{E}_{add}$
`SeqDot`	$\hat{E}_x$	$\hat{E}_y$	$\hat{E}_{mult}$	$\hat{E}_{add,seq}$
`SeqDotFMA`	$\hat{E}_x$	$\hat{E}_y$	0	$\hat{E}_{add,seq}$
`ParDot`	$\hat{E}_x$	$\hat{E}_y$	$\hat{E}_{mult}$	$\hat{E}_{add,par}$

the sequential dot-product `SeqDot` as a sum of partial rounding error components as

$$\hat{E}_{\texttt{SeqDot}} = \sum_{i=1}^{n} abu\epsilon_{x_i} + \sum_{i=1}^{n} abu\epsilon_{y_i} + \sum_{i=1}^{n} abu\epsilon_{P_i} + \sum_{i=1}^{n-1} B(\hat{S}_i^r)u\epsilon_{S_i},$$

where

- ❏ $\sum_{i=1}^{n} abu\epsilon_{x_i}$ represents the error component due to quantization error of x_i which is multiplied with y_i,
- ❏ $\sum_{i=1}^{n} abu\epsilon_{y_i}$ represents the error component due to quantization error of y_i which is multiplied with x_i,
- ❏ $\sum_{i=1}^{n} abu\epsilon_{P_i}$ is the rounding error component due to finite-precision multiplications $P_i = x_i y_i$,
- ❏ and $\sum_{i=1}^{n-1} B(\hat{S}_i^r)u\epsilon_{S_i}$ is the rounding error component due to finite-precision summations of P_i.

Therefore, it is straightforward to have a general AA expression $\hat{E}_{\texttt{Dot}}$ for the rounding error of any floating-point dot-product implementation as follows

$$\hat{E}_{\texttt{Dot}} = \hat{E}_x + \hat{E}_y + \hat{E}_{mult} + \hat{E}_{add}, \tag{5.1}$$

where $\hat{E}_x = \sum_{i=1}^{n} abu\epsilon_{x_i}$ and $\hat{E}_y = \sum_{i=1}^{n} abu\epsilon_{y_i}$ are the quantization error components, which are the same for all dot-product implementations. The error component due to multiplications $\hat{E}_{mult}$ and the error component due to additions $\hat{E}_{add}$ may, in general, be different depending on how the multiplications and additions are performed within different dot-product implementations. Computing dot-products using a fused multiply-accumulate, as for `SeqDotFMA`, can avoid the rounding error due to multiplications.

Parallel dot-products produce a smaller rounding error due to adding than sequential dot-products, which has been shown by Higham [5]. Sequential dot-products, i.e. `SeqDot` and `SeqDotFMA`, have the same AA expression $\hat{E}_{add,seq}$ for the error component of additions. Table 5.4 summaries the rounding error components for different dot-product implementations considered in this work.

To complete our analyses of rounding errors occurring in finite-precision computations of floating-point dot-products, we present the AA expressions for rounding error components of three different floating-point dot-product implementations in equations (5.2)-(5.4).

The AA expressions for the rounding error components of the sequential dot-products using basic operations and FMA are as follows:

$$\begin{aligned}\hat{E}_{\texttt{SeqDot}} &= \hat{E}_x + \hat{E}_y + \hat{E}_{mult} + \hat{E}_{add,seq} \\ &= \sum_{i=1}^{n} abu\epsilon_{x_i} + \sum_{i=1}^{n} abu\epsilon_{y_i} + \sum_{i=1}^{n} abu\epsilon_{P_i} + \sum_{i=1}^{n-1} B(\hat{S}_i^r)u\epsilon_{S_i},\end{aligned} \tag{5.2}$$

$$\begin{aligned}\hat{E}_{\texttt{SeqDotFMA}} &= \hat{E}_x + \hat{E}_y + \hat{E}_{add,seq} \\ &= \sum_{i=1}^{n} abu\epsilon_{x_i} + \sum_{i=1}^{n} abu\epsilon_{y_i} + \sum_{i=1}^{n-1} B(\hat{S}_i^r)u\epsilon_{S_i}. \end{aligned} \tag{5.3}$$

The AA expression for the rounding error components of the parallel dot-products having $\lceil log_2 n \rceil$ adding stages and $\lfloor n/2^k \rfloor$ additions performed at the *k-th* stage, while using basic operations, is represented as follows:

$$\begin{aligned}\hat{E}_{\texttt{ParDot}} &= \hat{E}_x + \hat{E}_y + \hat{E}_{mult} + \hat{E}_{add,par} \\ &= \sum_{i=1}^{n} abu\epsilon_{x_i} + \sum_{i=1}^{n} abu\epsilon_{y_i} + \sum_{i=1}^{n} abu\epsilon_{P_i} + \sum_{k=1}^{\lceil log_2 n \rceil} \sum_{j=1}^{\lfloor n/2^k \rfloor} B(\hat{S}_{k,j}^r)u\epsilon_{S_{k,j}}, \end{aligned} \tag{5.4}$$

Analytical expressions for AA probabilistic bounds of dot-products

After having derived the AA expressions for final rounding error of different dot-product implementations, we apply the AA probabilistic bounding operator (see Section 3.2.3), with considering all noise symbols as uniformly distributed random variables in [-1,+1], to compute the AA probabilistic bounds for the three different dot-product implementations. Corresponding analytical expressions are represented by equations (5.5), (5.6) and (5.7).

$$B_{\texttt{SeqDot}} = 2^{-p} K \frac{\sqrt{2}}{6} ab \cdot \sqrt{18n + K^2(n^2 + n - 2)}, \tag{5.5}$$

$$B_{\texttt{SeqDotFMA}} = 2^{-p} K \frac{\sqrt{2}}{6} ab \cdot \sqrt{12n + K^2(n^2 + n - 2)}, \tag{5.6}$$

$$B_{\texttt{ParDot}} = 2^{-p} K \frac{\sqrt{2}}{6} ab \cdot \sqrt{18n + 2K^2 \lceil log_2 n \rceil n}, \tag{5.7}$$

In these analytical equations, the constant K specifies the confidence interval for the AA probabilistic bound, where $K = 3$ corresponds to a confidence interval of 3σ (see Section 3.2.3). According to Figures 5.2 and 5.3, the error bounds for the two sequential dot-products using basic operations and FMA, i.e., `SeqDot` and `SeqDotFMA`, should be identical. The respective analytical expressions (5.5) and (5.6) for the error bounds are slightly different by small first-order terms of n, i.e., $18n$ and $12n$. Those differences are, however, negligible compared to the second-order terms n^2, especially for the large n, making the two analytical bounds in (5.5) and (5.6) almost identical.

Note that for large n, the expressions (5.5) and (5.6) grow approximately linearly with $\sqrt{n^2} = n$, whereas the expression (5.7) grows linearly with $\sqrt{n}$ (because $log_2 n$ grows much slower than any power of n). The term $log_2 n$ actually comes from the binary-tree structure of the parallel dot-product.

Verification of error bound correctness

We perform comparisons to verify the correctness of these analytical expressions with respect to the bounds evaluated by using our Matlab tool. Comparing the analytical AA probabilistic bounds given by equations (5.5), (5.6) and (5.7) with the AA probabilistic bounds evaluated by the `AAFloat` class results in maxmimum relative differences of 0.1% and 2.0% over the whole parameter space for sequential dot-products and the parallel dot-product, respectively, which verifies well our analytical expressions of the AA probabilistic bounds for different floating-point dot-product implementations.

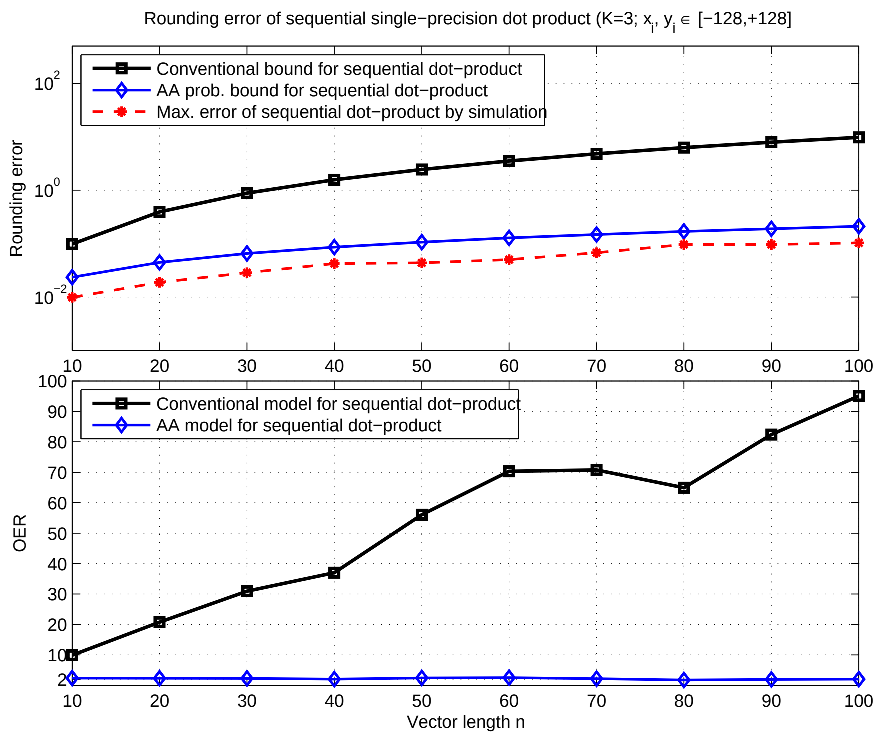


Figure 5.5 Rounding error of sequential single-precision dot-product and associated overestimation ratio *OER*

5.2.6 AA-based Error Analysis versus Conventional Error Analysis

In this section, we consider the usefulness of two error analysis techniques for bit width allocation for VLSI and reconfigurable systems design: Conventional error analysis versus AA based error analysis. For this investigation, we compare the conventional error bound with the AA probabilistic bound with respect to vector length n, while keeping the numerical range and working precision constant. Only the sequential dot-product implementation using basic operations is considered.

Conventional error analysis for floating-point dot-products has been presented in the literature (see [5] for a complete reference). The conventional (absolute) error bound of a sequential dot-product [5] is presented as

$$|\mathbf{x}^T\mathbf{y} - fl(\mathbf{x}^T\mathbf{y})| \leq \frac{nu}{1-nu}\sum_{i=1}^{n}|x_i y_i|,$$

where $u = 2^{-p}$ is the unit roundoff. We assume that $x_i \in [-a,+a], y_i \in [-b,+b]$ and use the maximal values of x_i, y_i to estimate the conventional bound, thereby $\sum_{i=1}^{n}|x_i y_i| \approx n \cdot ab$. For further simplification, we assume $1 - nu \approx 1$ (which is true when p is not too small and $n \leq 1000$). The conventional forward error bound for the sequential dot-product is, therefore, estimated as

$$|\mathbf{x}^T\mathbf{y} - fl(\mathbf{x}^T\mathbf{y})| \leq \quad 2^{-p} \cdot ab \cdot n^2. \tag{5.8}$$

The conventional forward error bound in (5.8) can be approximated as a function of n^2, while the AA probabilistic bound in (5.5) is estimated as a function of $\sqrt{n^2} = n$. Figure 5.5 shows the rounding error bounds and corresponding *OERs* of single-precision ($p = 24$) sequential dot-products derived by the conventional model in (5.8) and by the AA probabilistic bounding operator in (5.5). In theory, the *OERs* resulted in by the conventional model should be a function linear in n. In Figure 5.5, the conventional bound overestimates the realistic maximum error by a factor of 10 at vector length $n = 10$. The *OERs* of the conventional model then linearly increases with the vector length up to $n = 40$. There are small fluctuations in the *OERs* of the conventional model for the vector lengths $n = 50$ to $n = 80$, which seems to be due to the fluctuations in the simulations. The *OER* then continuously increases with increasing vector length and becomes even more pessimistic at vector length $n = 100$ with an *OER* of 95. Obviously, the conventional bound becomes useless for mantissa bit width allocation with the increase of vector length as the *OER* of this bound grows approximately linearly with vector length n.

The AA probabilistic bound for sequential dot-products is very close to the maximum rounding errors in practice (see Figure 5.5). More importantly, the *OER* obtained by AA modeling remains almost constant (i.e., fluctuating between 1 and 2, see lower subplot in Figure 5.5) over the whole range of vector lengths. This reveals that AA with a probabilistic bounding operator is able to provide tighter bounds than the conventional error model for floating-point error analysis.

5.2.7 Summary

We have shown that affine arithmetic with a probabilistic bounding operator is able to provide a tighter rounding error estimate compared to the bound provided by the conventional forward error analysis technique for different floating-point dot-product implementations over a wide range of parameters. Due to the tight bounds, the minimum mantissa bit width for hardware implementation can be determined and comparison of different dot-product architectures is possible. We have shown that the overall rounding error of a floating-point dot-product implementation can significantly be improved by employing a parallel architecture, rather than by employing an FMA. More importantly, the analytical rounding error models for all floating-point dot-product architectures are derived, allowing for an efficient design space exploration and which are key to specialised code generators.

As of now, the AA-based error model has been used for performing error analysis of simple floating-point arithmetic, i.e., addition, multiplication and FMA, via the example of the dot-product. In the remainder of this chapter, we will study the application of the AA-based error model in a much more complex algorithm for signal and speech processing: the Levinson-Durbin algorithm.

5.3 Rounding Error Analysis of Floating-Point Levinson-Durbin Algorithm

5.3.1 Motivation

Linear Prediction (LP) [6–8] is an important technique in signal and speech processing. This technique is based on autoregressive (AR) modeling and enables us to estimate the coefficients of AR filters and is thus closely related to the model of speech production. Suppose we are given a real sequence $1 = r_0, r_1, r_2, \ldots, r_n$, namely the auto-correlation coefficients of a wide-sense stationary discrete-time random process, that forms the special case of Yule-Walker equations of order n expressed in the expanded matrix form:

$$\begin{bmatrix} r_0 & r_1 & \cdots & r_{n-1} \\ r_1 & r_0 & \cdots & r_{n-2} \\ . & . & & . \\ . & . & & . \\ r_{n-1} & r_{n-2} & \cdots & r_0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ . \\ . \\ a_n \end{bmatrix} = - \begin{bmatrix} r_1 \\ r_2 \\ . \\ . \\ r_n \end{bmatrix}, \tag{5.9}$$

or in the compact matrix form

$$\mathbf{R} \cdot \mathbf{a} = -\mathbf{r}, \tag{5.10}$$

where $\mathbf{R}$ is the $n \times n$ square autocorrelation matrix, and the AR coefficients or predictor/filter coefficients $\mathbf{a}$ and the autocorrelation sequence $\mathbf{r}$ are $n \times 1$ column vectors.

The Levinson-Durbin algorithm [7, 8], which was suggested by Levinson (1947) and then reformulated by Durbin (1960), is an efficient algorithm for solving Yule-Walker equations like (5.10). The Levinson-Durbin algorithm belongs to a class of fast algorithms for solving Toeplitz systems whose computational complexity is $O(n^2)$ and storage requirement is $O(n)$. The underlying idea of the Levinson-Durbin algorithm can be described as follows. The algorithm starts from a known solution of the Yule-Walker equations with order $(n-1)$ leading to the solution of order n. In other words, the algorithm implicitly computes a factorization of the inverse of the Toeplitz matrix [65]. Naturally, the algorithm operates in an iterative manner. Thanks to its computational efficiency and low storage requirements, the Levinson-Durbin algorithm has played an important role not only in linear prediction but also in other applications involving Yule-Walker equations.

Other Levinson-like algorithms have been proposed to further benefit from the superior properties of the Toeplitz structure of the Yule-Walker equations. The split Levinson algorithm suggested by Delsarte and Genin [66] requires only half the amount of multiplications compared to the original algorithm. The Schur algorithm [67] (which was actually based on the Levinson-Durbin algorithm and implemented in fixed-point by Le Roux and Guegen in 1977) and the lattice algorithms [68] are other alternatives for determining the reflection coefficients of Yule-Walker equations. Similarly, split versions of the Schur and lattice algorithms were also proposed [69].

Beside the Levinson-Durbin algorithm and its improvements, other fast algorithms and asymptotically super-fast algorithms, i.e., with a computational complexity of $O(n \log n)$, for solving general Toeplitz systems having arbitrary righthand side vectors can be found in [65, 70, 71]. The numerical properties of those algorithms are generally poor (i.e., numerically unstable) and, therefore, those algorithms will not be considered in this work. The numerical properties of the Levinson-Durbin algorithm is presented later.

The common issue of Toeplitz solvers is that they suffer from the rounding effect of finite-precision computations. In contrast to other Toeplitz solvers, the Levinson-Durbin algorithm shows benefits, apart from its computational efficiency and low storage requirements, that make it an important Toeplitz solver. Firstly, the Levinson-Durbin algorithm computes both predictor coefficients and reflection coefficients in each iteration, thus the filter can be implemented alternatively in direct form or in lattice structure [8]. Secondly, the Levinson-Durbin algorithm provides directly results of all the subsystems (i.e., Yule-Walker equations with smaller orders than n) during the intermediate computations, thus the users can stop the computation at any step they want. The most important property of the Levinson-Durbin algorithm however is that it is numerically stable [30] for both fixed- and floating-point implementations.

Since the Levinson-Durbin algorithm is one of the most important Toeplitz solvers, efficient implementation for this algorithm is desirable. Generally, the Levinson-Durbin algorithm can be applied to any application that involves solving Yule-Walker equations. However, in practice, the most popular use of the Levinson-Durbin algorithm is its implementation in linear prediction and autoregressive modeling. Existing realizations can be classified into fixed-point implementations and floating-point implementations. While fixed-point hardware implementations of the Levinson-Durbin algorithm have been widely implemented in DSP [32, 33], reconfigurable hardware [34] and ASICs [31], floating-point arithmetic implementations of the Levinson-Durbin algorithm have, however, still not been thoroughly reported in existing literature. Another example of Levinson-Durbin implementation using rational arithmetic is presented in [35].

The goal of this work is the efficient implementation of a custom-precision floating-point Levinson-Durbin algorithm with respect to system order, hardware resources (area), performance and power consumption while meeting the accuracy requirements of the user. The critical question is how to determine the minimum bit width for the floating-point number format to guarantee the users' accuracy requirements. We aim at the optimization of a uniform floating-point bit width. Chapter 2 has shown that reduced minimum bit width of the floating-point number format leads to increased parallelism and performance for implementations on reconfigurable hardware. The benefit of reduced precision would still hold in floating-point applications like the Levinson-Durbin implementation. Therefore, minimum bit width allocation for the implementation of the Levinson-Durbin algorithm will result in minimal resource consumption and allow for improved performance.

In order to reach the goal, we apply affine arithmetic to estimate the numerical range and rounding error of floating-point operations in Levinson-Durbin iterations. We use the Matlab-based framework proposed in Chapter 4 for representation and evaluation of floating-point Levinson-Durbin computations. Both the ranges and rounding errors of all relevant quantities in the Levinson-Durbin implementation will be estimated. Those quantities are the reflection coefficients k_m, filter coefficients a_i and the short-term energy of the prediction error E_m. It has been shown in [30] that if the problem to be solved is already ill-conditioned then even an extremely stable algorithm can certainly result in totally wrong computed results. Therefore, we carefully take into account the condition number of the problem at hand when investigating the rounding effect of finite-precision computations of the Levinson-Durbin algorithm.

Since the Levinson-Durbin algorithm is used in the adaptation of the linear prediction coefficients, the accuracy of the resulting coefficients has some impacts on the accuracy of the synthesis filters in speech coding [7]. For subjective evaluation of the synthesis filter, the spectral distortion (SD) measure [7] is often used. Our ultimate concern is how the precision of the floating-point number format used in the Levinson-Durbin implementation affects the quality of the resulting synthesis filters.

In the remainder of this chapter, we will first describe the Levinson-Durbin algorithm and list some special properties of reflection coefficients (section 5.3.2) that are the basis for further studies of the algorithm in terms of rounding error bound estimation and bit width allocation. A summary of the numerical stability of the Levinson-Durbin algorithm is also presented and the normwise rounding error bound for filter coefficients is derived based on related work in [30]. Next we discuss the experimental setup for the custom-precision floating-point Levinson-Durbin implementation (section 5.3.3) and we present the experimental results on the rounding errors of the Levinson-Durbin algorithm that we observed when applying synthetic data and speech signals into the custom-precision floating-point Levinson-Durbin implementation (section 5.3.4). A central result of this thesis is to use the AA-based error analysis technique for deriving the AA-based rounding error bounds for filter coefficients and reflection coefficients in the Levinson-Durbin floating-point implementation as a function of the system order, precision and condition number of the Toeplitz matrix (section 5.3.5). Finally, we give our discussion on AA-based rounding error analysis and bit width allocation for floating-point Levinson-Durbin implementation.

5.3.2 Levinson-Durbin Algorithm

We give a description of the Levinson-Durbin (LD) algorithm and summarize the properties of the reflection coefficients. The specially restricted range of the reflection coefficients is then used to obtain a closed form estimate for the filter coefficients, providing a rough estimate for the range and rounding error of filter coefficients and the knowledge about the trend of the rounding error over iterations. The numerical stability of the LD algorithm is summarized based on related work.

Description of the Levinson-Durbin algorithm

The basic idea of the LD algorithm can be described as follows. The algorithm starts from a known solution $\mathbf{a}^{(m-1)}$ of the Yule-Walker equations with order $(m-1)$ and leads to the solution $\mathbf{a}^{(m)}$ of order m. Naturally, the algorithm proceeds in an iterative manner. In fact, the LD algorithm implicitly computes a factorization of the inverse of the Toeplitz matrix. This kind of inverse operation requires that all the submatrices factorized from the Toeplitz matrix have to be non-singular, or all the Toeplitz submatrices have to be positive definite.

More specifically, the LD algorithm computes the quantities k_m, E_m and $a_i^{(m)}$, $i = 1 \ldots m$, $m = 1 \ldots n$, by the initialization

$$E_0 = r_0, \quad a_0^{(0)} = 1 \tag{5.11}$$

and through the following iterations: For $m = 1$ to n,

$$\beta_m = \sum_{i=0}^{m-1} a_i^{(m-1)} r_{m-i} \tag{5.12}$$

$$k_m = -\frac{\beta_m}{E_{m-1}} \tag{5.13}$$

$$a_i^{(m)} = a_i^{(m-1)} + k_m a_{m-i}^{(m-1)} \quad (\forall\, i \in [1, m-1]), \quad a_0^{(m)} = 1, a_m^{(m)} = k_m \tag{5.14}$$

$$E_m = E_{m-1}\left(1 - k_m^2\right) \tag{5.15}$$

where k_m is the reflection coefficient at the m-th iteration of the LD algorithm; $a_i^{(m-1)}$ and $a_i^{(m)}$ are the i-th filter coefficients calculated at the $(m-1)$-th and the m-th iteration, respectively. At the m-th iteration, the LD algorithm generates a new reflection coefficient k_m and a short-term prediction error E_m, as well as the new set of filter coefficients $a_i^{(m)}$ from the previous filter coefficients $a_i^{(m-1)}$.

An alternative way for computing the short-term prediction error E_m is to use the autocorrelation coefficients r_i and the filter coefficients a_i at the m-th iteration as [7]

$$E_m = r_0 + \sum_{i=1}^{m} a_i^{(m)} r_i = \sum_{i=0}^{m} a_i^{(m)} r_i \quad (m = 1 \dots n), \quad E_0 = r_0 = 1. \tag{5.16}$$

The two equations (5.15) and (5.16) are mathematically equivalent. In this work, we use (5.15) for computing the short-term prediction error in the LD algorithm, both in experiments and in the AA-based error bound evaluation. The alternative equation (5.16) is more expensive than (5.15); it is therefore not used in practical implementations of the Levinson-Durbin algorithm. In this work, the equation (5.16) is only used later in subsection 5.3.5 to explain the possible range of the short-term prediction error in the AA-based error analysis of the LD algorithm.

Properties of reflection coefficients

Reflection coefficients have a number of interesting properties [7, 8, 72]:

- They are limited by one in magnitude, i.e., $|k_m| \leq 1$.[3] Note, that this is a consequence of using proper auto-correlation sequences as input to the LD algorithm and of assuming that an infinite precision floating-point number format is used. The LP-analysis filter, therefore, will have all its zeros inside or on the unit circle and the LP-synthesis filter will be stable (i.e., all poles are inside or on the unit circle). Therefore it can be concluded that the short-term energy E_m does not increase from iteration to iteration, i.e., $E_m = E_{m-1}(1 - k_m^2) \leq E_{m-1}$, while staying non-negative, i.e., $0 \leq E_m \leq E_0 = r_0 = 1$.

- Given the reflection coefficients k_m $(m = 1 \dots n)$, the filter coefficients a_i $(i = 1 \dots n, a_0 = 1)$ can be calculated by Equation (5.14).

- Given the filter coefficients a_i $(i = 1 \dots n, a_0 = 1)$, the reflection coefficients k_m $(m = 1 \dots n)$ can be computed directly as well (see [7] or [8]).

- There is a *one-to-one* correspondence between the two sets of quantities $\{E_0, k_1, k_2, \cdots k_n\}$ and $\{r_0, r_1, r_2, \cdots r_n\}$ in that if we are given the one, we may uniquely determine the other in an iterative manner [8].

- For (voiced) speech signals, it has been observed [72] that the reflection coefficients have a U-shaped probability density function, in which the reflection coefficients often concentrate near ± 1.

A closed form estimate for the range bound and error bound of filter coefficients

Based on the specially restricted property of the reflection coefficients within the range $[-1, 1]$, i.e., $|k_m| \leq 1$ $(m = 1 \dots n)$, and due to the fact that the filter coefficients a_i can be calculated directly

[3]The property that $|k_m| \leq 1$ can also be proven by using the two equations (5.12) and (5.16).

from reflection coefficients, the range bound and error bound for the filter coefficients a_i of order n can roughly be estimated by exploiting the relation in Equ. (5.14). In the following subsections, we assume all the reflection coefficients have been computed exactly without any rounding error.

Reflection-coefficient based range estimate for filter coefficients. We present a simple example calculating the filter coefficients a_i given the full set of reflection coefficients k_m using Equation (5.14) with a chosen order $n = 3$ as follows:

- Initialization ($m = 0$): $a_0^{(0)} = 1$
- Iteration $m = 1$, given k_1:

$$\begin{aligned} a_0^{(1)} &= 1 \\ a_1^{(1)} &= k_1 \end{aligned}$$

- Iteration $m = 2$, given k_2:

$$\begin{aligned} a_0^{(2)} &= 1 \\ a_1^{(2)} &= a_1^{(1)} + k_2 a_1^{(1)} = k_1 + k_2 k_1 \\ a_2^{(2)} &= k_2 \end{aligned}$$

- Iteration $m = 3$, given k_3:

$$\begin{aligned} a_0^{(3)} &= 1 \\ a_1^{(3)} &= a_1^{(2)} + k_3 a_2^{(2)} = k_1 + k_2 k_1 + k_3 k_2 \\ a_2^{(3)} &= a_2^{(2)} + k_3 a_1^{(2)} = k_2 + k_3(k_1 + k_2 k_1) = k_2 + k_3 k_1 + k_3 k_2 k_1 \\ a_3^{(3)} &= k_3 \end{aligned}$$

This example shows that the *i-th* filter coefficient at the iteration m, i.e., $a_i^{(m)}$ $(i = 0, 1 \ldots m;\ m = 0, 1 \ldots n)$, is computed as a finite sum of the products of reflection coefficients. The most important implication in this example is that the upper bound for the product of the reflection coefficients is restricted by $|\prod k_m| \le 1$ because each partial coefficient $|k_m| \le 1$. Therefore, the upper bound for the filter coefficient $a_i^{(m)}$ is exactly equal to the number of products of the reflection coefficients.

In fact, Cybenko [30] showed that the upper bound for the coefficient $a_i^{(m)}$ computed at iteration m of the LD algorithm is the binomial coefficient of the x^i term in the polynomial expansion of a binomial power $(1 + x)^m$. If we define $R_{a_i^{(m)}}$ as the upper bound for the filter coefficient a_i computed at the m-th iteration of the LD algorithm, then this upper bound is estimated as follows:

$$R_{a_i^{(m)}} = \binom{m}{i} = \frac{m!}{i!(m-i)!}, \tag{5.17}$$

where $m!$ denotes the factorial of m, i.e., $m! = 1 \times 2 \times \cdots \times m$ and $0! = 1$.

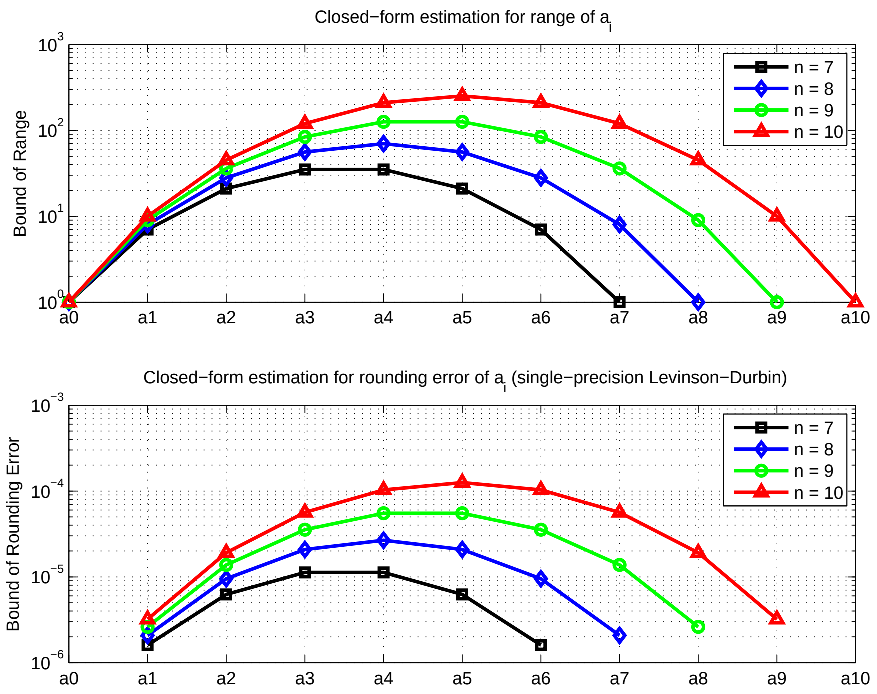


Figure 5.6 A closed-form estimate (based on the reflection-coefficients) for the range and rounding error of filter coefficients a_i for the single-precision Levinson-Durbin algorithm

Reflection-coefficient based error estimate for filter coefficients. As mentioned earlier in Chapter 3, the rounding error of a floating-point operand is directly proportional to the size of the range of that floating-point operand. Using the conventional rounding error model of floating-point number suggested by Higham [5] along with the upper bound of the range of filter coefficient a_i in (5.17), an estimate of the floating-point rounding error bound for filter coefficient a_i computed at iteration m of the LD algorithm is given in (5.18)

$$E_{a_i^{(m)}} \quad = \quad E_{a_i^{(m-1)}} + E_{a_{m-i}^{(m-1)}} + R_{a_i^{(m)}} \cdot 2^{-p}, \tag{5.18}$$

where $E_{a_i^{(m)}}$ is the upper bound of the rounding error of $a_i^{(m)}$, and p is the precision. Note that by convention, $E_{a_0^{(m)}} = 0$ and $E_{a_m^{(m)}} = 0$, $\forall m \in \{0, 1, \dots n\}$ since $a_0^{(m)} = 1$ and $a_m^{(m)} = k_m$ (which is assumed to be computed exactly).

Discussion. The upper bound for the range of filter coefficients estimated by (5.17) is a precise estimate and therefore it can be used as the reference to evaluate the accuracy of the ranges estimated by AA-based error analysis technique which will be presented later in Section 5.3.5.

Figure 5.6 illustrates the range bounds estimated by (5.17) and the error bounds estimated by (5.18) for the filter coefficients a_i corresponding to some system orders. The numerical range of each filter coefficient a_i increases with the system order n. Given a system order n, the ranges of coefficients a_i are even-symmetric via the center coefficient. As the size of the rounding error of a floating-point number is directly proportional to the range of that number, the even-symmetry property of the range of the filter coefficients suggests that the respective rounding error bound of a_i should follow a similar trend versus the coefficient a_i, confirmed by the lower subplot of Figure 5.6.

For the rounding error estimation of the filter coefficients, we are aware that equation (5.18) is based on the assumption that all the reflection coefficients k_m are strictly bounded and accurately computed in the range $[-1, 1]$ with no rounding error. In other words, the estimate in (5.18) does not capture the accumulating effect of rounding errors in the reflection coefficients and, therefore, it is a less precise estimate. However, this rough estimate is important because it provides knowledge about the trend of the rounding error of filter coefficients over iterations that is very important for understanding the behavior of the resulting rounding error in filter coefficients in the LD iterations for practical applications, which will be presented in the experimental results in Section 5.3.4.

The numerical stability of the Levinson-Durbin algorithm

The numerical stability of the LD iteration has been a subject of several publications in the literature [30, 73]. It has been shown in [30] that the LD algorithm is numerically stable for both fixed- and floating-point implementations. The size of the rounding errors of computed quantities in the LD algorithm, however, strongly depends on the condition number of the Toeplitz matrix being solved. This section presents the rounding error bound for the filter coefficients taking into account the accumulation of rounding errors over iterations of the LD algorithm and the condition number of the Toeplitz system at hand. For doing so, we extend the calculation by Cybenko [30] for the bound of the residual of the righthand side vector $\mathbf{r}$ in (5.10) to represent the error bound of the filter coefficient vector $\mathbf{a}$ as a function of system order n, precision p and the condition number.

The condition number [5] of the Toeplitz matrix $\mathbf{R}$ is defined as

$$\kappa(\mathbf{R}) \quad = \quad ||\mathbf{R}|| \cdot ||\mathbf{R}^{-1}|| \tag{5.19}$$

where $||\cdot||$ is some matrix norm. An in-depth discussion about vector-norm and matrix-norm can be found in textbook, like [5]. In accordance with the work of Cybenko [30], the 1-norm will be considered in this thesis. The definition for other norms and condition numbers can be found in Appendix C. The 1-norms for an $n \times n$ square matrix $\mathbf{R}$ and for an $n \times 1$ vector $\mathbf{a}$ are defined as

$$||\mathbf{R}||_1 \quad = \quad \max_{1 \le j \le n} \sum_{i=1}^{n} |r_{ij}| \tag{5.20}$$

$$||\mathbf{a}||_1 \quad = \quad \sum_{i=1}^{n} |a_i| \tag{5.21}$$

and, therefore, the 1-norm for a matrix in (5.20) is also called "max column sum".

Since $r_0 = 1$ and $\mathbf{R}$ is positive definite and an autocorrelation matrix, i.e., $|r_i| \le 1, \quad i = 1 \ldots n$, we obtain

$$1 \le ||\mathbf{R}||_1 \le n. \tag{5.22}$$

As a consequence, the size of $||\mathbf{R}^{-1}||_1$ will essentially define the size of the condition number $\kappa(\mathbf{R})$ of the Toeplitz system. Given all reflection coefficients k_m and r_0, the lower and upper bounds for $||\mathbf{R}^{-1}||_1$ have been determined in [30], from which the lower and upper bounds of the condition number of the Toeplitz system can be calculated.

In contrast to [30], in this work we are to estimate the bound for the range and rounding error of all the reflection coefficients k_m and filter coefficients a_i computed by the floating-point LD implementation, the reflection coefficients k_m are, therefore, not known in advance. However, we assume that the condition number $\kappa(\mathbf{R})$ of the problem at hand is given, such that the size of $||\mathbf{R}^{-1}||_1$ can then be estimated and used to derive the upper bound of the absolute error for the filter coefficients a_i. Since $||\mathbf{R}||_1$ is bounded by (5.22), the size of $||\mathbf{R}^{-1}||_1$ will essentially determine the condition number of the autocorrelation matrix and can be estimated as

$$||\mathbf{R}^{-1}||_1 \approx \kappa(\mathbf{R}). \tag{5.23}$$

Now we present the upper bound of the absolute rounding error of filter coefficients a_i by using the bound for the residual given in Cybenko's work [30]. Cybenko derived the bound for the residual in a floating-point implementation of the LD algorithm as follows. Suppose that $\mathbf{a}$ and $\hat{\mathbf{a}}$ are the true solution and finite-precision computed solution of the Yule-Walker equations, respectively. The computed solution is then

$$\hat{\mathbf{a}} = \mathbf{a} + \boldsymbol{\alpha},$$

where $\boldsymbol{\alpha}$ is the perturbation vector or absolute error vector (of length n) due to the accumulation of rounding errors, and n is the order of the Yule-Walker equation. Obviously, this computed solution will satisfy a perturbed system of equations

$$\begin{aligned} \mathbf{R}\hat{\mathbf{a}} &= -\hat{\mathbf{r}} \\ \mathbf{R}(\mathbf{a} + \boldsymbol{\alpha}) &= -\hat{\mathbf{r}} \\ \mathbf{R}\mathbf{a} + \mathbf{R}\boldsymbol{\alpha} &= -\mathbf{r} + \boldsymbol{\delta} \end{aligned}$$

The vector $\boldsymbol{\delta}$ (of length n) is called the residual vector and satisfies

$$\mathbf{R}\boldsymbol{\alpha} = \boldsymbol{\delta}. \tag{5.24}$$

The upper bound of the residual vector for a floating-point implementation is given as [30]

$$||\boldsymbol{\delta}||_1 \leq u\left(\frac{n^2}{2} + 11n\right)\left[\prod_{j=1}^{n}(1 + |k_j|) - 1\right] + O(u^2) \tag{5.25}$$

where u is the unit roundoff (i.e., $u = 2^{-p}$) depending on the precision p. The term $O(u^2)$ briefly expresses the fact that only first order errors are considered important in the analysis. For numerical evaluation of the bound of the residual, the term $O(u^2)$ will not be considered and the reflection coefficients take their maximum value $|k_j| = 1$.

At this point, we are ready to determine the bound for the absolute error vector according to Cybenko's derivation. The absolute error vector $\boldsymbol{\alpha}$ is defined from (5.24) as

$$\boldsymbol{\alpha} = \mathbf{R}^{-1}\boldsymbol{\delta}. \tag{5.26}$$

Applying the 1-norm onto (5.26) gives

$$||\boldsymbol{\alpha}||_1 = ||\mathbf{R}^{-1}\boldsymbol{\delta}||_1,$$

thus the upper bound for the 1-norm of error vector $\boldsymbol{\alpha}$ is determined as (see relation (C.8))

$$||\boldsymbol{\alpha}||_1 \leq ||\mathbf{R}^{-1}||_1 \cdot ||\boldsymbol{\delta}||_1, \tag{5.27}$$

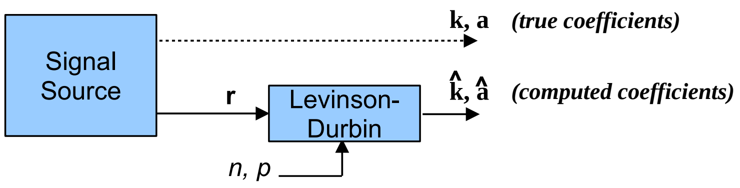


Figure 5.7 General setup for rounding error evaluation of custom-precision floating-point Levinson-Durbin implementation

where $||\boldsymbol{\delta}||_1$ is computed by (5.25), and $||\mathbf{R}^{-1}||_1$ is approximated via the condition number by (5.23) With these settings, the upper bound for the 1-norm of error vector $\boldsymbol{\alpha}$ is explicitly computed as a function of condition number $\kappa(\mathbf{R})$, precision p (or $u = 2^{-p}$) and system order n as

$$||\boldsymbol{\alpha}||_1 = \kappa(\mathbf{R}) \cdot u \cdot \left(\frac{n^2}{2} + 11n\right) \cdot (2^n - 1), \tag{5.28}$$

showing that the upper bound for the 1-norm of the error vector linearly scales with the condition number $\kappa(\mathbf{R})$ of the Toeplitz system. In fact, the experimental data in Section 5.3.4 will confirm this conclusion that the rounding error of the filter and reflection coefficients in the LD iterations increases linearly with the condition number of the Toeplitz system at hand.

5.3.3 Experimental Setup

This section presents our setup to experimentally estimate the rounding errors of the coefficients a_i and k_m computed by the custom-precision floating-point LD implementation as a function of filter order n, precision p (number format), condition number of the Toeplitz system and signal types. The experimental results will be presented in the next section 5.3.4.

In the following, the general setup for a rounding error evaluation of the LD implementation is first presented, focusing on the discussion of choosing the true coefficients extracted from the signal source and computed coefficients given by the LD implementation. Then the measures used for error evaluation of the LD implementation are defined. Finally, two particular setups for a synthetic data scenario and a speech signal scenario are described in detail.

General setup

Figure 5.7 describes the general setup for rounding error evaluation of the custom-precision LD implementation. Given the precision p and system-order n, the auto-correlation vector $\mathbf{r}$ is applied as the input to the custom-precision LD implementation in Matlab. The signal source can be a real-world autoregressive signal from nature or it can be synthesized for simulation purposes. For speech processing, which is the focus of this work, the rounding errors are studied with two types of signal sources: speech signals and synthetic data. Synthetic data is randomly generated taking into consideration the distribution of the reflection coefficients. For speech data, the well-known TIMIT [74] speech database is used.

To evaluate the rounding errors, the computed reflection coefficient vector $\hat{\mathbf{k}}$ and filter coefficient vector $\hat{\mathbf{a}}$ in the p-precision number format are compared with the corresponding true coefficient vectors $\mathbf{k}$ and $\mathbf{a}$. The estimated coefficients $\hat{\mathbf{k}}$ and $\hat{\mathbf{a}}$ are computed by the custom-precision LD implementation that receives the autocorrelation input vector $\mathbf{r}$ estimated from the signal source.

How should the true values $\mathbf{k}$ and $\mathbf{a}$ be chosen and which number format should be used for those true value representations?

In terms of numerical representation, we choose the double-precision number format ($p = 53$) in Matlab to represent the true coefficient vectors $\mathbf{k}$ and $\mathbf{a}$. The choice of true values for $\mathbf{k}$ and $\mathbf{a}$ depends on particular experimental scenarios. The same setup is applied to the auto-correlation vector $\mathbf{r}$. The vector $\mathbf{r}$ is represented in double-precision and calculated depending on particular cases of the experimental setup. Different experimental setup scenarios will be described in more detail in the remainder of this section.

The experimental error evaluation is performed in a frame-based manner. Given the system order n and precision p, a sufficiently large number of data frames, 10^5 frames for synthetic data and 116395 ($\approx 10^5$) frames for speech data, are processed with the LD implementation. For each frame, the autocorrelation sequence $\mathbf{r}$ is computed in double-precision and used as input to the LD implementation. Statistical measures for the rounding errors are estimated taking into consideration the condition number κ of all the data frames processed.

Custom-precision floating-point LD implementation. Custom-precision floating-point arithmetic operations are implemented in Matlab using MPFR as described in section 4.3. The Matlab code of the function `levcus` implementing the custom-precision floating-point LD algorithm is presented in Appendix D.

Measure definition for rounding error evaluation of Levinson-Durbin implementation

In the following, we define the necessary measures for rounding error evaluation. These measures include component-wise and norm-wise errors. Taking into account the condition number of the system, we can evaluate the maximum error corresponding to a specified condition number.

In terms of speech processing, we present the average spectral distortion as an objective measure for evaluating the quality of a linear predictor versus the number format used.

Component-wise errors. For floating-point computation the most useful measures of the accuracy are the absolute error and the relative error. For the LD algorithm, the component-wise absolute and relative errors are defined for the computed filter coefficient $\hat{a}_i$ as

$$e_{abs,\hat{a}_i} = |a_i - \hat{a}_i|,$$

$$e_{rel,\hat{a}_i} = \frac{|a_i - \hat{a}_i|}{|a_i|},$$

and for the computed reflection coefficient $\hat{k}_i$ as

$$e_{abs,\hat{k}_i} = |k_i - \hat{k}_i|,$$

$$e_{rel,\hat{k}_i} = \frac{|k_i - \hat{k}_i|}{|k_i|}.$$

Note that the relative error is defined only for $a_i \neq 0$ and $k_i \neq 0$.

Since the filter coefficients and reflection coefficients are represented by two vectors $\hat{\mathbf{a}}$ and $\hat{\mathbf{k}}$, the corresponding error measures are preferably represented in vector notation. The absolute error vectors for filter coefficients and reflection coefficients corresponding to one data frame are

$$\begin{aligned}
\mathbf{e}_{abs,\hat{\mathbf{a}}} &= |\mathbf{a}-\hat{\mathbf{a}}| = [|a_1-\hat{a}_1|, |a_2-\hat{a}_2|, \ldots, |a_n-\hat{a}_n|]^T, && (5.29)\\
\mathbf{e}_{abs,\hat{\mathbf{k}}} &= |\mathbf{k}-\hat{\mathbf{k}}| = [|k_1-\hat{k}_1|, |k_2-\hat{k}_2|, \ldots, |k_n-\hat{k}_n|]^T. && (5.30)
\end{aligned}$$

By definition $a_0 = 1$, thus the rounding error of a_0 is zero and not included in the error vectors.

Norm-wise errors. For rounding error evaluation the norm applied onto these error vectors is used. Different vector norms exist. This work uses the 1-norm (see Appendix C) because it takes into account all error components and it is also used in related work by Cybenko [30]. For one data frame, the norms of the absolute error and relative error of the predictor and reflection coefficients are defined by

$$||\mathbf{e}_{abs,\hat{\mathbf{a}}}||_1 = ||\mathbf{a}-\hat{\mathbf{a}}||_1 = \sum_{i=1}^{n} |a_i-\hat{a}_i|, \tag{5.31}$$

$$||\mathbf{e}_{rel,\hat{\mathbf{a}}}||_1 = \frac{||\mathbf{a}-\hat{\mathbf{a}}||_1}{||\mathbf{a}||_1} = \frac{\sum_{i=1}^{n}|a_i-\hat{a}_i|}{\sum_{i=1}^{n}|a_i|}, \tag{5.32}$$

$$||\mathbf{e}_{abs,\hat{\mathbf{k}}}||_1 = ||\mathbf{k}-\hat{\mathbf{k}}||_1 = \sum_{i=1}^{n} |k_i-\hat{k}_i|, \tag{5.33}$$

$$||\mathbf{e}_{rel,\hat{\mathbf{k}}}||_1 = \frac{||\mathbf{k}-\hat{\mathbf{k}}||_1}{||\mathbf{k}||_1} = \frac{\sum_{i=1}^{n}|k_i-\hat{k}_i|}{\sum_{i=1}^{n}|k_i|}, \tag{5.34}$$

where n is the system order, $\hat{\mathbf{a}}$ and $\hat{\mathbf{k}}$ are the coefficient vectors computed at the p-precision number format.

Maximum errors with respect to condition number. As the rounding errors of the coefficients in the LD algorithm depend on the condition number of the Toeplitz matrix, as already shown in 5.3.2, we need to take into account the condition number when evaluating the rounding errors. We denote $\mathbb{D}$ the set of all data frames investigated and N the number of data frames ($N = 10^5$ for synthetic data, $N = 116395$ for speech data extracted from TIMIT database). Obviously the set $\mathbb{D}$ has N elements. We use the 1-norm to compute the condition number for each data frame. The maximum absolute and relative errors over N frames can be calculated as

$$\max_{\mathbb{D}} ||\mathbf{e}_{abs,\hat{\mathbf{a}}}||_1,\ \max_{\mathbb{D}} ||\mathbf{e}_{rel,\hat{\mathbf{a}}}||_1 \quad \text{for filter coefficients;}$$

$$\max_{\mathbb{D}} ||\mathbf{e}_{abs,\hat{\mathbf{k}}}||_1,\ \max_{\mathbb{D}} ||\mathbf{e}_{rel,\hat{\mathbf{k}}}||_1 \quad \text{for reflection coefficients.}$$

Now taking into account the condition number $\kappa(\mathbf{R})$ of the Toeplitz matrix corresponding to each data frame, we can extract the maximum error for $\hat{\mathbf{a}}$ and $\hat{\mathbf{k}}$ for all the frames having a condition number less than a specified upper bound. We denote $\mathbb{D}_{101}$ the set of all the data frames having the condition number less than or equal to 10^1. Obviously, $\mathbb{D}_{101}$ has less than or equal to N data frames. The maximum absolute and relative errors for the data frames having the condition number $\kappa(\mathbf{R}) \leq 10^1$ are defined by

$$\max_{\mathbb{D}_{101}} ||\mathbf{e}_{abs,\hat{\mathbf{a}}}||_1,\ \max_{\mathbb{D}_{101}} ||\mathbf{e}_{rel,\hat{\mathbf{a}}}||_1 \quad \text{for filter coefficients;}$$

$$\max_{\mathbb{D}_{101}} ||\mathbf{e}_{abs,\hat{\mathbf{k}}}||_1, \; \max_{\mathbb{D}_{101}} ||\mathbf{e}_{rel,\hat{\mathbf{k}}}||_1 \quad \text{for reflection coefficients,}$$

i.e., the maximum errors are evaluated in the subset $\mathbb{D}_{101}$. Similarly we can define $\mathbb{D}_{102}$, $\mathbb{D}_{103}$, etc. as the sets of all the data frames having $\kappa(\mathbf{R}) \leq 10^2$, $\kappa(\mathbf{R}) \leq 10^3$, etc. Similar estimates for the maximum errors of these new sets can also be made. The higher condition number the data frames have, the larger the resulting errors will be.

Spectral distortion measure. In parametric *vocoders* (*vo*ice *coders*), the number format used to quantize the predictor and reflection coefficients directly affects the quality of the encoded speech signal. In order to evaluate the quality of an LPC (*linear predictive coding*) vocoder, an objective measure or a distance measure, which is preferably independent of the chosen filter structure, is required [7]. A common measure is the *mean spectral distortion.*

The spectral distortion measure for a single speech frame is defined as follows [7]:

$$SD \;=\; \sqrt{\frac{1}{2\pi}\int_{-\pi}^{\pi}\Big[10\log_{10}|H(e^{j\Omega})|^2 - 10\log_{10}|\hat{H}(e^{j\Omega})|^2\Big]^2 \mathrm{d}\Omega} \qquad \text{[dB]}, \tag{5.35}$$

where $H(e^{j\Omega})$ and $\hat{H}(e^{j\Omega})$ are the frequency responses of the synthesis filters corresponding to the true or unquantized coefficients represented in an infinite precision number format and the computed or quantized coefficients represented in a p-precision number format, respectively. The mean spectral distortion $\overline{SD}$ is evaluated by averaging over all the speech frames in the test data.

For LPC vocoders, it is commonly accepted that a mean value $\overline{SD} \leq 1$ dB corresponds to transparent speech quality [7]. According to [75], "transparent" quantization of LPC parameters means that the LPC quantization does not introduce any additional audible distortion in the coded speech; i.e., the two versions of coded speech - the one obtained by using unquantized LPC parameters and the other by using the quantized LPC parameters - are indistinguishable through listening.

Before computing the spectral distortion measure, we would like to show the relation between the frequency responses, $H(e^{j\Omega})$ and $\hat{H}(e^{j\Omega})$, of the synthesis filters and the predictor coefficients a_i and $\hat{a}_i$ computed in infinite-precision and in p-precision, respectively. In speech processing, the vocal tract is modeled by an all-pole filter. This all-pole filter is reconstructed by the synthesis filter at the receiving side of a digital speech transmission system. The predictor coefficients a_i characterize the frequency response $H(e^{j\Omega})$ or $H(z)$ (in the z domain) of the synthesis filter as follows

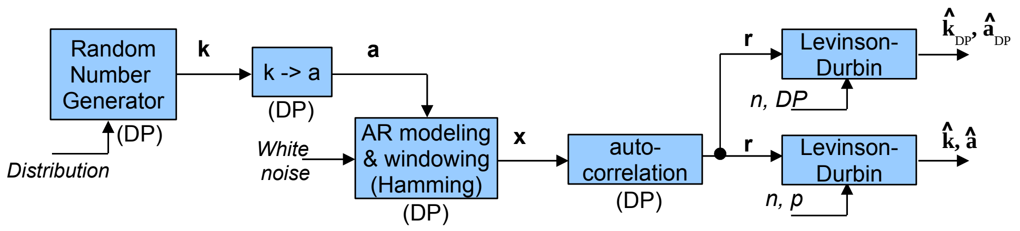
$$H(z) \;=\; \frac{1}{A(z)} = \frac{1}{1+\sum_{i=1}^{n} a_i z^{-i}}, \tag{5.36}$$

where $A(z) = 1 + \sum_{i=1}^{n} a_i z^{-i} = 1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}$ is the frequency response of the LP-analysis filter of order n at the transmitting side. Similarly, the predictor coefficients $\hat{a}_i$ estimated in finite-precision arithmetic characterize the frequency response $\hat{H}(z)$ as

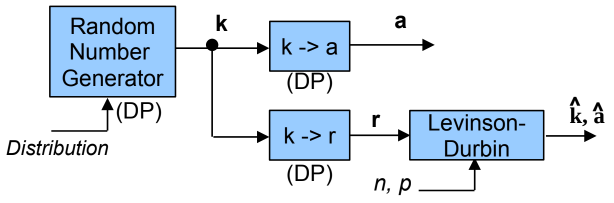
$$\hat{H}(z) \;=\; \frac{1}{\hat{A}(z)} = \frac{1}{1+\sum_{i=1}^{n} \hat{a}_i z^{-i}}. \tag{5.37}$$

We are now to compute the spectral distortion measure. Computing the spectral distortion directly from its definition in (5.35) is expensive because it requires the computation of an integral. Alternatively, the spectral distortion can be computed using the real cepstral coefficients as follows [7]

$$SD \;=\; 20\log_{10}(e)\sqrt{2}\sqrt{\sum_{i=1}^{n}(cc_i - \widehat{cc}_i)^2} \qquad \text{[dB]}, \tag{5.38}$$



a) Waveform-based Parameter Estimation



b) Parameter Conversion

Figure 5.8 Experimental setup with synthetic data for custom-precision Levinson-Durbin implementation in Matlab

in which $e \approx 2.71828$ is the base of the natural logarithm, cc_i and $\widehat{cc}_i$ are the real cepstral coefficients that correspond to the two systems $H(z)$ and $\hat{H}(z)$, respectively, and n is the filter order. Furthermore, for an all-pole model, the real cepstral coefficients cc_i can be computed from the predictor coefficients a_i by the following recursion

$$cc_i \quad = \quad a_i + \sum_{m=1}^{i-1} \frac{i-m}{i} cc_{i-m} a_m, \qquad 1 \leq i \leq n. \tag{5.39}$$

The transform from predictor coefficients $\hat{a}_i$ to cepstral coefficients $\widehat{cc}_i$ can be made similarly. The detailed derivation for the relations in (5.38) and (5.39) is presented in [7, Section 3.7].

As mentioned earlier in the general experimental setup, the coefficients a_i are computed in the double-precision number format and the coefficients $\hat{a}_i$ are computed in p-precision number format. The cepstral coefficients cc_i and $\widehat{cc}_i$ and the spectral distortion SD in (5.38) are evaluated using the double-precision number format.

To summarize, we will evaluate the spectral distortion measure as a function of the precision of the floating-point number format used for custom-precision LD implementation by using the real cepstral coefficients, which, in turn, are calculated from the predictor coefficients, as shown in relations (5.38) and (5.39), respectively.

Setup for synthetic data

For the generation of synthetic data, we start with the reflection coefficient vector $\mathbf{k}$ being generated in the range $[-1, +1]$ to guarantee the stability of the autoregressive models considered. Figure 5.8 describes the block diagrams of two setups that can be used for the generation of $\mathbf{r}$ from $\mathbf{k}$:

- *Waveform-based parameter estimation* that estimates the autocorrelation coefficient vector $\mathbf{r}$

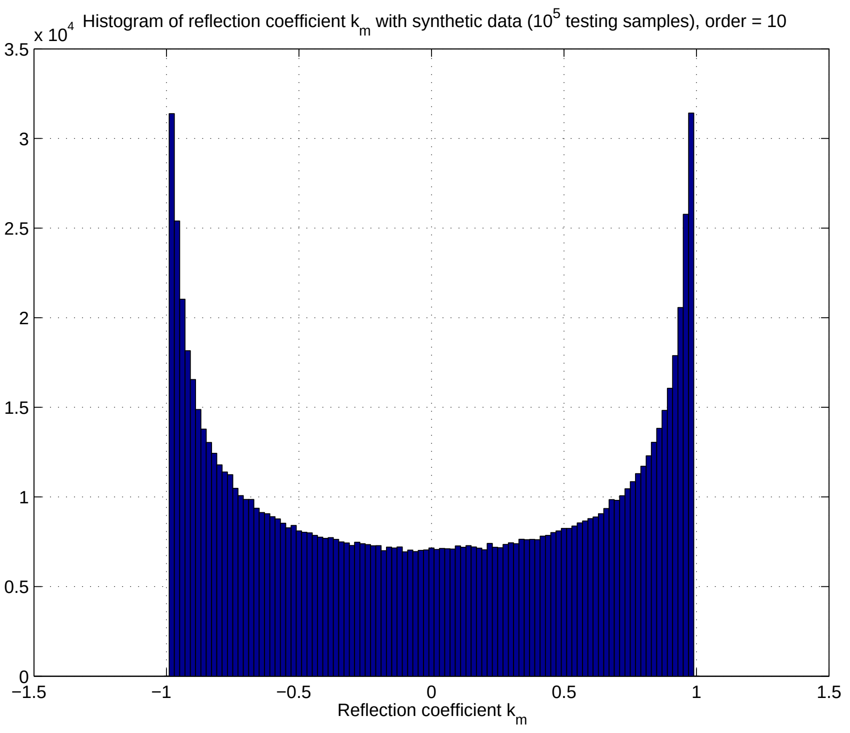


Figure 5.9 Histogram of reflection coefficients for synthetic data

from the waveform of autoregressive sequences $\mathbf{x}$. The autoregressive sequence $\mathbf{x}$ is, in turn, a windowed output of an autoregressive model characterized by coefficients $\mathbf{a}$ and $\mathbf{k}$.

❑ *Parameter conversion* that computes the autocorrelation coefficient vector $\mathbf{r}$ directly from the reflection coefficient vector $\mathbf{k}$ [8, Section 3.4].

For numerical experiments, the distribution of k can be chosen arbitrarily. For speech processing, it has been shown [72] that the reflection coefficients are often concentrated around ±1 and that the probability density distribution of k_m usually has a U-shaped distribution for (voiced) speech signals. For generation of coefficients k_m satisfying the usually-observed distribution of speech signals and restricted to $(-1, +1)$, the cosine function can be used. A cosine function $k = cos(w)$ of an angle w uniformly distributed in $[0, 2\pi]$ has the desired distribution as illustrated by Figure 5.9. Random generation of the reflection coefficient vector $\mathbf{k}$ using the cosine function is used for both the waveform-based parameter estimation and parameter conversion setups as shown in Figure 5.8. For both setups, the filter coefficient vector $\mathbf{a}$ is computed from the reflection coefficient vector $\mathbf{k}$ using the LD algorithm with the `rc2poly` function in Matlab. The difference between the two experimental setups is the later use of the filter coefficient vector $\mathbf{a}$.

Waveform-based parameter estimation. For the *waveform-based parameter estimation setup* (Figure 5.8a), $\mathbf{a}$ is used as the input coefficients of an AR model excited by white noise to generate the time series $\mathbf{x}$, which will then be used to compute the autocorrelation vector $\mathbf{r}$ by using the `xcorr` function in Matlab. Given the filter order n and precision p of the floating-point number format, two

LD implementations are performed, resulting in two sets of computed reflection coefficients and filter coefficients: $\hat{\mathbf{k}}$ and $\hat{\mathbf{a}}$ in the p precision number format, and $\hat{\mathbf{k}}_{DP}$ and $\hat{\mathbf{a}}_{DP}$ in double-precision ($p = 53$).

The reason for running the LD algorithm both in double-precision and in p-precision can be explained as follows. For numerical simulations using the waveform-based parameter estimation setup, the computed autocorrelation vector $\mathbf{r}$ cannot accurately characterize the original model represented by $\mathbf{k}$ and $\mathbf{a}$ because the time series $\mathbf{x}$ is only a windowed version of the true infinite-length sequence due to memory and simulation time restriction. Therefore, the quantities $\hat{\mathbf{k}}_{DP}$ and $\hat{\mathbf{a}}_{DP}$, even computed at very high precision, are not necessarily identical to the original $\mathbf{k}$ and $\mathbf{a}$. We call this the *finite-length window effect*. The more samples used for generation of $\mathbf{x}$, the closer the computed coefficients are to the original coefficients, but the larger memory required and the longer simulation times are needed, i.e., $\hat{\mathbf{k}}_{DP} - \mathbf{k} \to 0$ for number of samples $\to \infty$.

If the LD implementation is performed in the p-precision number format, computed reflection coefficients and filter coefficients are also suffering from rounding errors. Generally speaking, in the waveform-based parameter estimation setup, the computed quantities $\hat{\mathbf{k}}$ and $\hat{\mathbf{a}}$ suffer numerical errors from both the finite-length window effect and the finite-precision computation effect, while the computed quantities $\hat{\mathbf{k}}_{DP}$ and $\hat{\mathbf{a}}_{DP}$ suffer only from the finite-length window effect. From Figure 5.8a, it is quite reasonable to state that both double-precision and p-precision LD implementations are affected by the same amount of finite-length window effect errors. This is extremely useful as it allows for the separate estimation of the errors due to the finite-length window effect and the finite-precision computation effect. The error due to the finite-length window effect is the difference between $\mathbf{k}$, $\mathbf{a}$ and $\hat{\mathbf{k}}_{DP}$, $\hat{\mathbf{a}}_{DP}$. The error due to the finite-precision computation effect is the difference between $\hat{\mathbf{k}}_{DP}$, $\hat{\mathbf{a}}_{DP}$ and $\hat{\mathbf{k}}$, $\hat{\mathbf{a}}$.

For the autocorrelation computation block in Figure 5.8a, signal $\mathbf{x}$ is multiplied with a Hamming window before it is used to calculate the autocorrelation sequence $\mathbf{r}$. The autocorrelation sequence $\mathbf{r}$ is computed with the Matlab function `xcorr` using its own raw and unscaled implementation. The autocorrelation sequence is then normalized by the zero-lag coefficient r_0 (i.e., $r_0 = 1$ after normalization).

Note that for the waveform based parameter estimation setup for synthetic data, there may be another setup scenario in which the autoregressive signal $\mathbf{x}$ is computed directly from randomly generated filter coefficients $\mathbf{a}$. This scenario does not start from the reflection coefficients $\mathbf{k}$ but starts directly from a random generator for the filter coefficients $\mathbf{a}$, corresponding to a path from $\mathbf{a}$ to $\mathbf{x}$ then to $\mathbf{r}$ in Figure 5.8a. However, one should not use this scenario because when starting from randomly generated filter coefficients $\mathbf{a}$ it cannot be guaranteed that all the zeros of the linear-predictive analysis filter are inside the unit circle, and, therefore, this can result in unstable linear-predictive synthesis filters. While this instability might even be tolerable over the duration of a short frame, the stability guarantees of linear prediction analysis using the LD algorithm will result in prediction coefficient estimates corresponding to a stable filter, thereby generating a systematic bias between the prediction coefficients used for data synthesis and the coefficients formed in the subsequent analysis step.

Parameter conversion. We are only interested in the rounding error due to finite-precision computation and, therefore, the error due to the finite-length window effect should be removed. For doing so, the parameter conversion setup can be used.

In the *parameter conversion setup* (Figure 5.8b), the autocorrelation vector $\mathbf{r}$ is directly computed from the reflection coefficients and the zero-lag autocorrelation coefficient $r_0 = 1$. This is due to

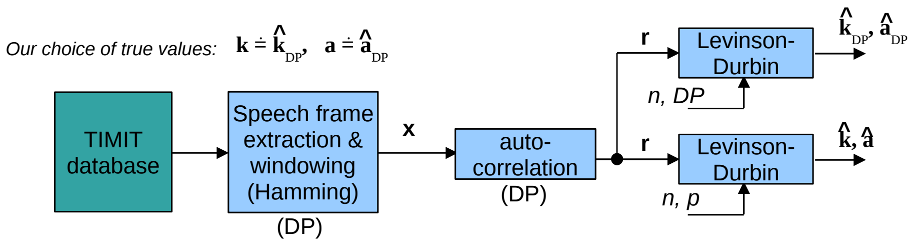

Our choice of true values: $\mathbf{k} \doteq \hat{\mathbf{k}}_{DP}$, $\mathbf{a} \doteq \hat{\mathbf{a}}_{DP}$
TIMIT database
Speech frame extraction & windowing (Hamming)
(DP)
x
auto-correlation
(DP)
r
Levinson-Durbin
n, DP
$\hat{\mathbf{k}}_{DP}$, $\hat{\mathbf{a}}_{DP}$
r
Levinson-Durbin
n, p
$\hat{\mathbf{k}}$, $\hat{\mathbf{a}}$

Figure 5.10 Experimental setup with speech signals for custom-precision Levinson-Durbin implementation in Matlab

one of the useful properties of the reflection coefficients computed with the LD algorithm. With the parameter conversion setup, the effect of the finite-length window can be avoided. Besides, the rounding errors due to the floating-point computation of the parameter conversion itself (i.e., from $\mathbf{k}$ to $\mathbf{a}$ and from $\mathbf{k}$ to $\mathbf{r}$), as performed in double-precision, are negligible compared to the rounding errors due to the finite-precision LD iterations. Therefore, in this case, given the filter order n and precision p, the original quantities $\mathbf{k}$ and $\mathbf{a}$ are used as references to estimate the absolute and relative errors of the finite-precision LD implementation in p precision.

For floating-point error analysis with synthetic data, which is the focus of this thesis, the *parameter conversion setup* will be used.

Setup for speech signals

For speech signals, we use the `TIMIT` speech corpus. The block diagram for custom-precision LD implementation using speech signals is shown in Figure 5.10. The speech signals have been recorded at a sampling rate of 16 kHz (when processing with the LD implementation, we keep the original sampling rate) and the speech frame duration is chosen as 20 miliseconds, resulting in an amount of 320 samples per frame. The speech frame overlapping factor is 50%.

Similar to the waveform-based parameter estimation for synthetic data, two LD implementations are performed with double-precision and p-precision number formats, respectively. Because each speech frame consists of only 320 samples, the computed reflection coefficients and filter coefficients will suffer from the finite-length window effect. Since the original reflection and filter coefficients characterizing the human vocal tract are unknown, the finite-length window effect error cannot be estimated. Again, it is quite reasonable to state, from Figure 5.10, that both double-precision and p-precision LD implementations are affected by the same amount of error due to the finite-length window effect, making their difference equal to the finite-precision rounding error. Therefore, for experiments using speech data, double-precision quantities are used as references to evaluate the absolute and relative errors of the LD implementation in p-precision format. In other words, our choice for the true values of filter and reflection coefficients are: $\mathbf{a} = \hat{\mathbf{a}}_{DP}$ and $\mathbf{k} = \hat{\mathbf{k}}_{DP}$.

Again, for the autocorrelation computation blocks in Figure 5.10, signal $\mathbf{x}$ is multiplied with a Hamming window before it is used to calculate the autocorrelation sequence $\mathbf{r}$. Similarly to the waveform-based parameter estimation setup, the autocorrelation sequence $\mathbf{r}$ in speech signal setup is computed with the Matlab function `xcorr` using its own raw and unscaled implementation and is then normalized by the zero-lag coefficient, such that $r_0 = 1$.

The histogram of the reflection coefficients obtained from all the voiced and unvoiced speech frames

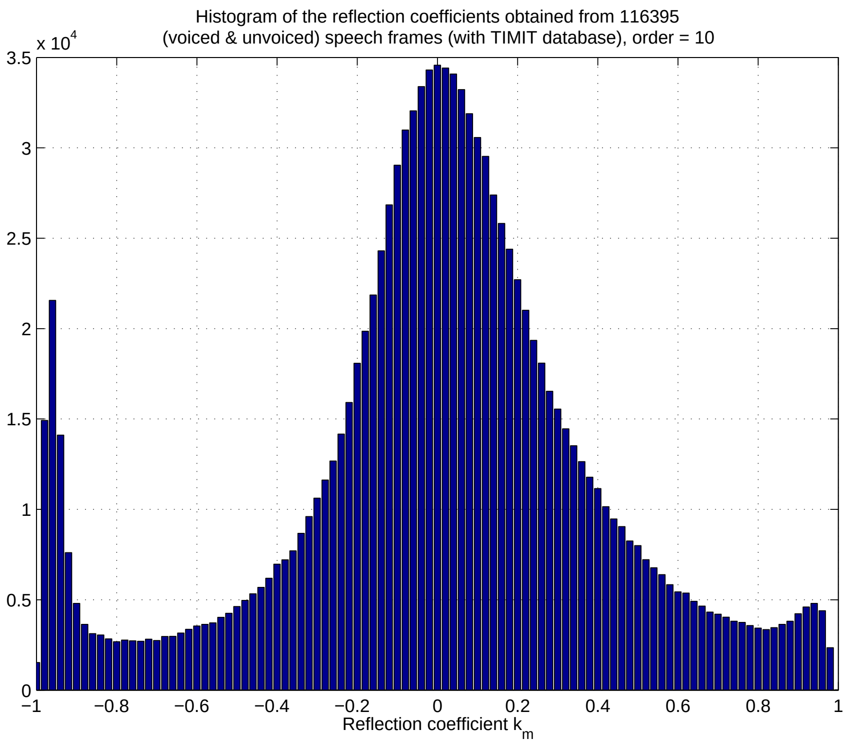


Figure 5.11 Histogram of reflection coefficients for speech (TIMIT)

with TIMIT database is shown in Figure 5.11. As a consequence of this mixture, the resulting distribution of the reflection coefficients does not perfectly match a U-shaped distribution for voiced speech signals. The central region with the small values of reflection coefficients (i.e., $-0.2 \leq k_m \leq 0.2$) seems to be due to the unvoiced/silence speech frames, while the peak around -1 and the small peak around $+1$ are probably due to the voiced speech frames.

5.3.4 Experimental Results

This section presents the experimental rounding error of filter coefficients a_i and reflection coefficients k_m of the custom-precision floating-point LD implementation (see preceding section 5.3.3 for the experimental setup). In general, the rounding error of the LD algorithm is a function of multiple input parameters including: signal sources (synthetic data or speech data), working precision, filter order and the condition number of the Toeplitz matrix at hand. Obviously, evaluating the experimental error is, therefore, a complex task. We, however, try to simplify this task while still guaranteeing the completeness of presentation via the following settings:

- The experimental errors from synthetic data and speech signals (TIMIT database) are reported.
- Both errors of filter coefficients a_i and reflection coefficients k_m are presented in component-wise and norm-wise representations (see Section 5.3.3 for component-wise and norm-wise error measure definitions).
- For the component-wise representation, we are interested in the partial rounding error of each coefficient (a_i or k_m) as well as the accumulating trend of the rounding error from the current

Table 5.5 Relative frequency of the condition number κ (at order $n = 10$) for speech (116395 frames) and synthetic data (10^5 frames)

Condition number	Speech	Synthetic data
$\kappa \le 10^1$	4.14 %	0 %
$\kappa \le 10^2$	24.95 %	1.41 %
$\kappa \le 10^3$	40.73 %	21.20 %
$\kappa \le 10^4$	60.61 %	65.53 %
$\kappa \le 10^5$	90.52 %	92.91 %
$\kappa \le 10^6$	99.93 %	99.33 %
$\kappa \le 10^7$	100 %	99.97 %

iteration to the next iteration given a specific system order, and therefore we fix the system order at $n = 10$. The experimental component-wise rounding errors for the filter coefficients and reflection coefficients are shown in Figure 5.12 and Figure 5.13, respectively.

- For the norm-wise representation, we represent the total rounding error in the filter coefficient vector or the reflection coefficient vector as a single representative norm value and we are interested in how this norm value will change when varying the working precision and the condition number of the Toeplitz matrix. The respective experimental rounding errors for the filter coefficients and reflection coefficients are shown in Figure 5.14 and Figure 5.15.
- All the errors displayed in Figures 5.12-5.15 are the maximum errors.
- Three values for the condition number[4] $\kappa = 10^4, 10^5, 10^6$ are considered, as we observed that up to 60% of speech frames and 65% of synthetic data frames among all the investigated frames have a condition number $\kappa \le 10^4$, and up to 99% of frames have a condition number $\kappa \le 10^6$ (see Table 5.5).

In addition, the mean spectral distortion $\overline{SD}$ for the speech signals and synthetic data versus the precision of the floating-point number format used in the LD implementation at order $n = 10$ is presented in Figure 5.16.

Rounding error for filter coefficients

The component-wise absolute error of filter coefficients a_i at order $n = 10$ is shown in Figure 5.12. The maximum error for each filter coefficient a_i, corresponding to the condition numbers $\kappa = 10^4, 10^5, 10^6$, is reported. The experimental data in Figure 5.12 shows that the rounding errors of the filter coefficients are symmetric. The central filter coefficients a_5 and a_6 suffer from the largest effect of rounding errors. With respect to the condition number, it is obvious from Figure 5.12 that the rounding error of filter coefficients increases by an order of magnitude when the condition number increases by an order of magnitude. In other words, the experimental rounding error in the LD iterations scales linearly with the condition number, confirming the numerical property of the LD algorithm earlier mentioned in section 5.3.2. The synthetic data frames result in higher rounding errors in comparison with the speech frames, which is explainable as the synthetic data frames generate higher condition numbers.

[4]In accordance with Cybenko's work [30], we compute the condition number κ of the Toeplitz matrix using the 1-norm.

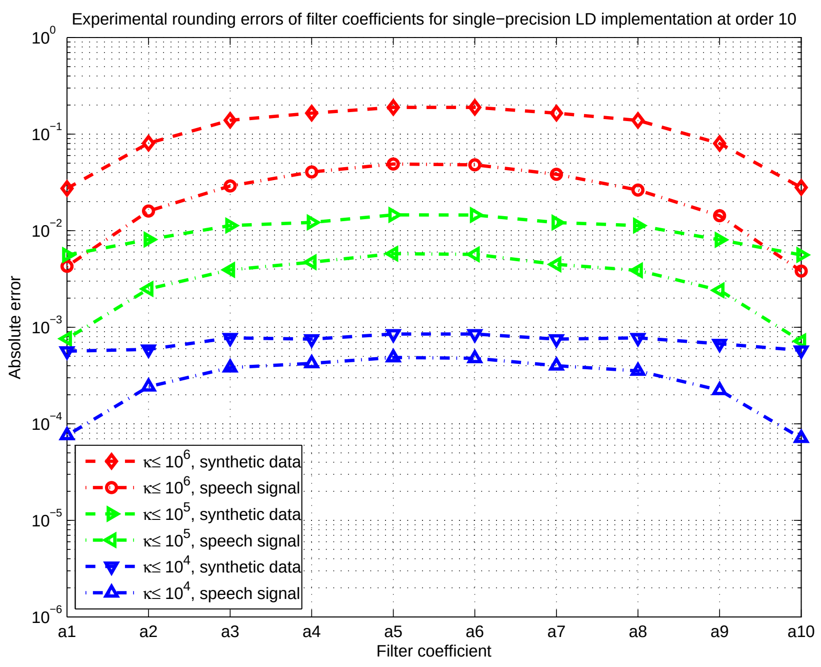


Figure 5.12 Experimental rounding errors of filter coefficients for single-precision Levinson-Durbin implementation at order 10 versus the condition number κ

The trend of the experimental errors for filter coefficient a_i is similar to the trend of an earlier error bound estimate reported in Figure 5.6 in subsection 5.3.2.

Rounding error for reflection coefficients

The component-wise rounding errors for the reflection coefficients are shown in Figure 5.13. For speech data, the rounding error increases dramatically at the beginning of the LD iterations, then it remains almost unchanged at later iterations. For synthetic data, a similar trend happens when starting the iterations.

Rounding error versus number format

The ultimate goal of rounding error analysis presented in this work is to perform optimal bit width allocation. It is therefore desirable to know how the rounding error changes when the working precision (or number format) changes.

Keeping the system order at $n = 10$ and increasing the precision from $p = 20$ to $p = 30$ in 2-bit steps, we can investigate the error of filter coefficients a_i and reflection coefficients k_i versus precision and condition number. A norm-wise error representation is considered. Experimental results are depicted in Figure 5.14 and Figure 5.15 for filter coefficients and reflection coefficients, respectively, as a function of precision and condition number (i.e., $\kappa = 10^4, 10^5, 10^6$) of the Toeplitz matrix.

For floating-point arithmetic, the rounding error of a single operation is inversely proportional to the precision. Therefore, increasing the precision by 1 mantissa bit would decrease the rounding error

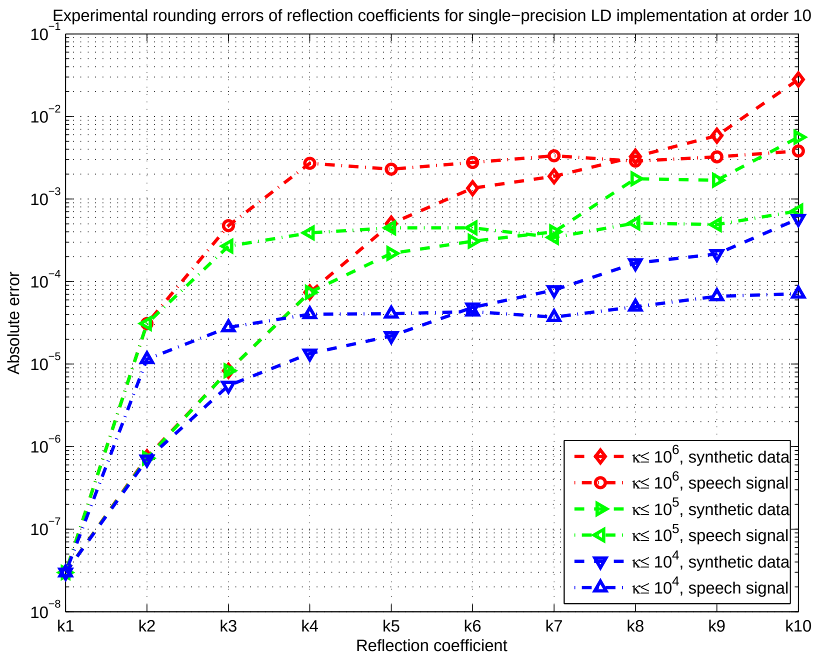


Figure 5.13 Experimental rounding errors of reflection coefficients for single-precision Levinson-Durbin implementation at order 10 versus the condition number κ

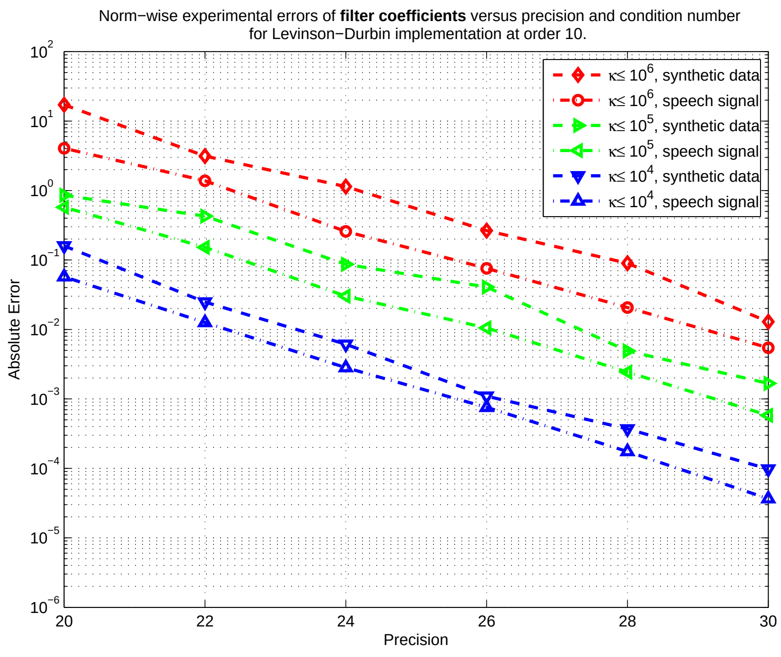


Figure 5.14 Norm-wise error of filter coefficients a_i versus precision and condition number (system order $n = 10$)

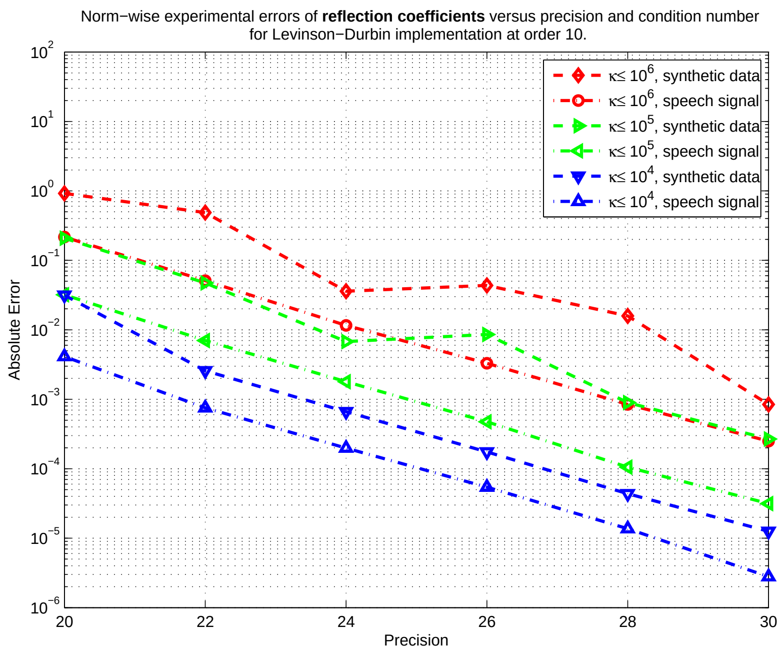


Figure 5.15 Norm-wise error of reflection coefficients k_i versus precision and condition number (system order $n = 10$)

by about 2 times, which is mostly verified for the experimental errors of both filter coefficients and reflection coefficients in Figure 5.14 and Figure 5.15.

In terms of condition number, it is easy to observe that when increasing the condition number by an order of magnitude, the rounding error also increases by an order of magnitude at the same precision.

Spectral distortion versus number format

We estimate the mean spectral distortion as a function of the precision used. Although the mean spectral distortion is commonly used in speech processing, we present here the mean spectral distortion for the synthetic data as well. The mean values are evaluated over 116395 speech frames for the speech signal and over 10^5 data frames for the synthetic data. Before computing the spectral distortion, all the frames corresponding to unstable synthesis filters (i.e., the frames corresponding to reflection coefficients $|k_m| > 1$) are removed. The instability is caused by the finite-precision computation of the LD algorithm in reduced precision. For the speech signals, up to 25% of the speech frames were removed at the precision $p = 10$. For the synthetic data, the number of removed frames was much more with a ratio of up to 50% at the precision $p = 10$. Note that at precision $p = 10$, we have the unit roundoff $u = 2^{-10} \approx 10^{-3}$, which heuristically shows that the solution of the Yule-Walker equation may not be defined if the condition number exceeds 10^{+3} [64].

Figure 5.16 presents the mean spectral distortion for speech and synthetic data versus the precision of the floating-point number format used in the LD implementation of order $n = 10$. For speech, there is almost no distortion for higher precisions, e.g., from 20 to 24 mantissa bits; at $p = 24$ bits, the mean spectral distortion $\overline{SD} = 0.001$ dB. Starting from the precision $p = 17$ bits, the spectral distortion for speech increases constantly with decreasing precision at a rate of about 0.3 dB per mantissa bit.

For the synthetic data, the spectral distortion increases dramatically with reduced precision and is

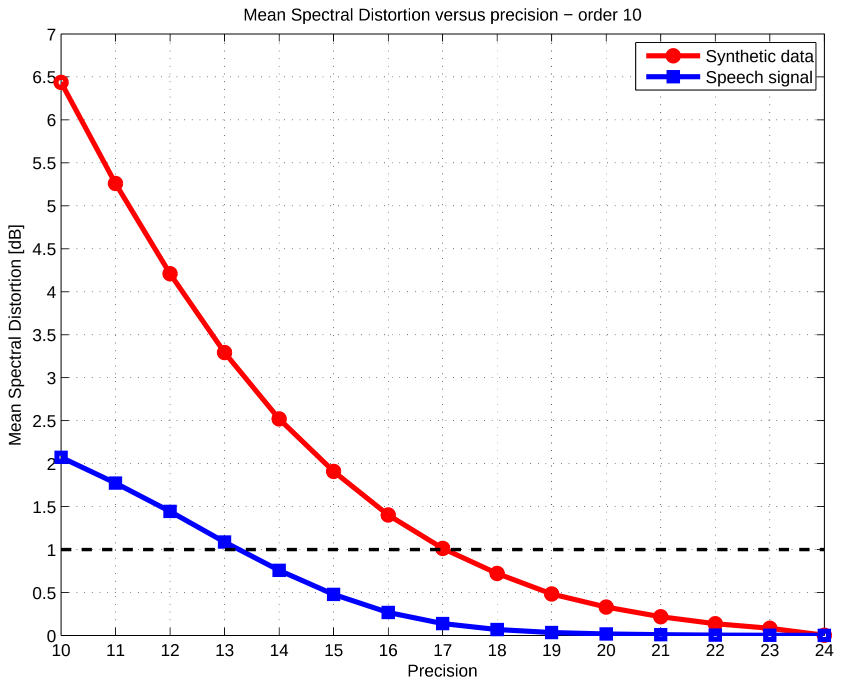


Figure 5.16 Mean spectral distortion versus precision (order 10); the upper bound for transparent speech quality is 1 dB

much worse compared to the respective mean value for speech at the same working precision (e.g., 0.002 dB at $p = 24$ and 6.5 dB at $p = 10$), probably due to the higher condition numbers of the Toeplitz systems generated from the synthetic data, reported in Table 5.5.

The upper bound 1 dB for transparent speech quality may suggest that the floating-point LP-analysis filter and the LD implementation at the transmitting side may be implemented using 14 or 13 mantissa bits. In fact, after calculation in the LP-analysis filter, the LPC parameters need to be quantized before they are transmitted. The quantization generates quantization noise, further degrading the quality of the synthesized speech at the receiving side. For guaranteeing the transparency of the synthesized speech, a 16 mantissa-bit floating-point number format, corresponding to an $\overline{SD} = 0.3$ dB, should be used. This choice seems to be reasonable as today many fixed-point implementations work with 16 bits, which seems to be adequate for speech.

5.3.5 AA-based Error Analysis for Levinson-Durbin Algorithm

In the following, we will investigate the numerical range and rounding error bounds of the LD algorithm based on AA model using the `AAFloat` tool in Matlab. Our goal is to understand the applicability of the AA error model and the `AAFloat` class in real applications like the LD iterations.

Introduction

The Yule-Walker equations

$$\mathbf{R} \cdot \mathbf{a} = -\mathbf{r},$$

where $\mathbf{R}$ is the autocorrelation matrix, which is an $n \times n$ positive definite and symmetric Toeplitz matrix, and the right-hand side autocorrelation vector $\mathbf{r}$ is closely related to $\mathbf{R}$, can be solved efficiently by using the LD algorithm. The input to the LD iterations is the autocorrelation sequence $\mathbf{r0} = [r_0, r_1, r_2 \ldots, r_n]^T = [1; \mathbf{r}]$, where $r_0 = 1$ and $\mathbf{r} = [r_1, r_2 \ldots, r_n]^T$. Generally, the autocorrelation coefficient r_i could be in the range $[-1, +1]$. For speech processing at a sampling rate of 8 kHz, the order of the linear predictor is often $n = 8$ to $n = 13$.

As earlier mentioned in subsection 4.4.2, the evaluation of the reflection coefficient k_m in the LD algorithm (Equ. (5.11)-(5.15)) involves taking the inverse of the prediction error E_{m-1} computed at the previous iteration, thereby possibly leading to a division by zero if the range of the prediction error E_{m-1} contains zero. In other words, the division has a major impact on the possible range and rounding error of the reflection coefficients and, as a consequence, on the range and rounding error of the filter coefficients (and via propagation on the reflection coefficients and filter coefficients of the subsequent iterations). To what extent this affects the estimated bound depends on the respective input data. Recall that at the m-th iteration the prediction error E_m can also be estimated via the autocorrelation coefficients and the filter coefficients using equation (5.16) in subsection 5.3.2 as

$$E_m = r_0 + \sum_{i=1}^{m} a_i^{(m)} r_i = \sum_{i=0}^{m} a_i^{(m)} r_i,$$

which clearly shows that both the autocorrelation coefficients r_i and the system order will affect the prediction error E_m in the sense that the range of the prediction error may contain zero (which leads to a division by zero) by increasing the range of r_i and/or increasing the system order.

The efficiency and ease of use of any given error model and its respective software tool depend on how well it can model the algorithm at hand and how well it can handle the special cases (e.g., division by zero when having very small values for the prediction error E_m, ultimately approaching zero). In case of executing a division by zero, the `AAFloat` tool provides the user two options to continue the execution of the algorithm at hand: using a default setting or specifying a user's lower bound (see section 4.2.3). These options will be applied for the AA-based error estimation of the LD algorithm.

Cases and scenarios considered. In the remainder of this section, we will investigate in more detail different scenarios for AA-based error evaluation of the LD algorithm. Table 5.6 describes five scenarios of performing AA-based error analysis of the Levinson-Durbin algorithm by using the `AAFloat` tool. These five (5) scenarios can be classified into three (3) main cases as follows:

- ❑ *Case 1: Restricted range for input parameters.* This case includes *scenario 1*, where the ranges of the autocorrelation coefficient r_i and the system order n are restricted. The range of the prediction error, therefore, does not contain zero over the AA-based evaluation of the LD algorithm. Hence, the AA-based evaluation is executed without any division by zero over all the iterations.

- ❑ *Case 2: General range for input parameters and using default setting of the `AAFloat` tool.* The range of the prediction error contains zero if the ranges of the autocorrelation coefficient r_i and/or the system order n are unrestricted. The division by zero happens and the `AAFloat` tool uses the default setting for the smaller bound in magnitude. This case is demonstrated via *scenario 2* and *scenario 3.*

- ❑ *Case 3: General range for input parameters and using user's setting.* The autocorrelation coefficient r_i and the system order n have general values, similar to case 2. The range of the

Table 5.6 Different scenarios of the `AAFloat` tool for AA-based error analysis of the Levinson-Durbin algorithm at $p = 24$

Name	Range of r_i	Order n	Handling division by zero by
Scenario 1	$[0.17,\ 0.23]$	is fixed at $n = 10$	no division by zero
Scenario 2	$[0.17,\ 0.23]$	is increased from $n = 10 \dots 15$	using default setting of the tool
Scenario 3	$[-1,\ 1]$	is increased from $n = 1 \dots 5$	using default setting of the tool
Scenario 4	$[0.17,\ 0.23]$	is increased from $n = 10 \dots 15$	specifying users' range
Scenario 5	$[-1,\ 1]$	is increased from $n = 1 \dots 5$	specifying users' range

prediction error contains zero. The division by zero occurs and the user specifies the smaller bound in magnitude for the prediction error to be used by the `AAFloat` tool. This case includes *scenario 4* and *scenario 5*.

Case 1: Restricted range for input parameters

In the first scenario, we start the investigation of the LD algorithm with a system order $n = 10$ and an intentionally chosen range $[0.17,\ 0.23]$ for the autocorrelation coefficients r_i. A parameter set of restricted range and system order is chosen to make sure the range of the prediction error will not contain zero during the AA-based error evaluation. In fact, we observe that Fang also used a specific 10×10 square matrix in [4] when applying AA for performing range analysis of the Cholesky decomposition.

We use the `AAFloat` class to perform the AA-based error analysis of the LD algorithm in Matlab by constructing a simple autocorrelation matrix $\hat{\mathbf{R}}$ having each autocorrelation coefficient as $\hat{r}_0 = 1$, $\hat{r}_i = 0.2 + 0.03\epsilon_i$, $\epsilon_i \in [-1, +1]$ and $i = 1, 2, \dots, n$. For example, the Yule-Walker equations of order 3 are represented in an AA representation as follows

$$\begin{bmatrix} 1 & (0.2 + 0.03\epsilon_1) & (0.2 + 0.03\epsilon_2) \\ (0.2 + 0.03\epsilon_1) & 1 & (0.2 + 0.03\epsilon_1) \\ (0.2 + 0.03\epsilon_2) & (0.2 + 0.03\epsilon_1) & 1 \end{bmatrix} \begin{bmatrix} \hat{a}_1 \\ \hat{a}_2 \\ \hat{a}_3 \end{bmatrix} = - \begin{bmatrix} (0.2 + 0.03\epsilon_1) \\ (0.2 + 0.03\epsilon_2) \\ (0.2 + 0.03\epsilon_3) \end{bmatrix}. \tag{5.40}$$

In this example, each autocorrelation coefficient is in the range $[0.17,\ 0.23]$, i.e., $r_i \in [0.17,\ 0.23]$. The system order and the working precision are chosen as $n = 10$ and $p = 24$ (i.e., single-precision), respectively.

For evaluating the accuracy of the AA-based model applied for the LD algorithm, we compare the range and error bounds of the quantities in the LD algorithm, estimated by the AA-based model, with the range and rounding error reported by running simulations of 10^5 data frames. For each data frame, the autocorrelation coefficients r_i are uniformly generated in the range $[0.17,\ 0.23]$. For each data frame, the rounding error due to simulations is computed as the difference between the result of the 24-bit LD implementation and the result of the double-precision (53-bit) LD implementation. The maximum rounding error among 10^5 data frames is used for comparing with the AA-based error bound. The Matlab code for the custom-precision Levinson-Durbin implementation using the MPFR functions as .MEX files is reported in Appendix D.1. Refer to subsection 5.3.3 for an extensive discussion of the experimental setup for the custom-precision implementation of the LD algorithm.

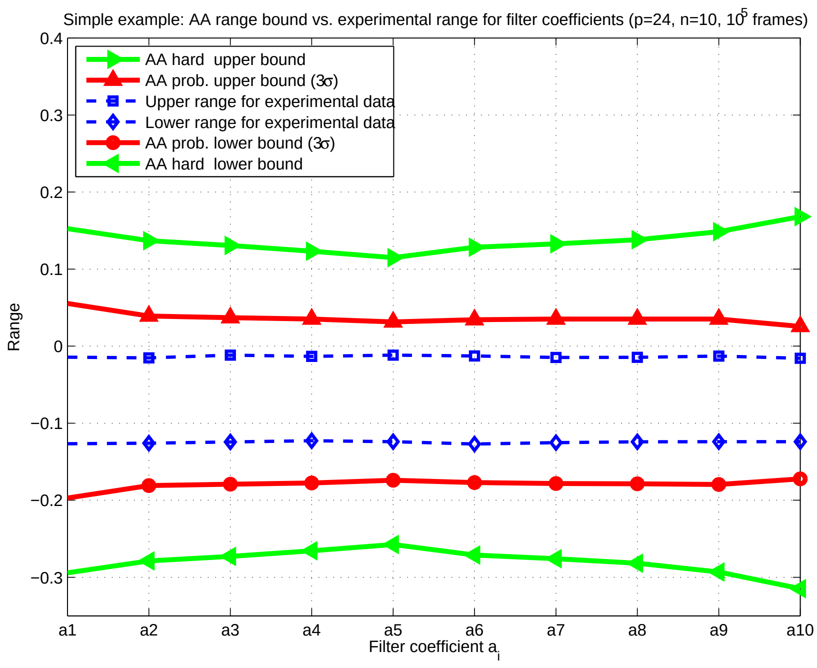


Figure 5.17 The range bound for filter coefficients a_i for the single-precision Levinson-Durbin algorithm at $n = 10$

For the AA-based error analysis of the LD algorithm, both the hard and probability bounding operators are used. The computational expressions for the LD algorithm have been presented in subsection 5.3.2. The Matlab code for AA-based error analysis of the LD algorithm using the `AAFloat` class is listed in Appendix D.2.

Figures 5.17 and 5.18 present the range estimation for the filter and reflection coefficients of the LD algorithm in comparison with the experimental results, respectively. Both the lower and upper bounds for the coefficients are reported. The two figures show that affine arithmetic is able to model quite accurately the range of the coefficients in the LD algorithm. For a quantitative comparison, we compute the overestimation ratio for the range. We use the width of the range, rather than the lower or upper bound only, to evaluate the accuracy of the AA-based model. The overestimation ratio for the range is the ratio of the width of the range bound over the width of the experimental results. The maximum overestimation ratios for the range of the coefficients in figures 5.17 and 5.18 are reported in Table 5.7, showing that, in this example, the affine arithmetic using a probabilistic bounding operator can reliably model the range bound for the LD algorithm. Refer to Appendix E.1 (iterations 1 to 10) for a full report of scenario 1.

The shapes of the AA-based hard range bounds for the filter coefficients and the reflection coefficients (i.e., the green curves) in Figures 5.17 and 5.18 are surprising for us. For the filter coefficients (Figure 5.17), the AA-based hard range (lower and upper) bounds in the special and restricted range (of input parameters) seems to have the inverse trend of the closed-form estimates based on the reflection coefficients in the general case presented earlier in subsection 5.3.2 (Figure 5.6). For this special and restricted range of input parameters, we have no further explanation for the trend of the filter coefficients. For the reflection coefficients, we observe a strong divergence for the AA-based hard

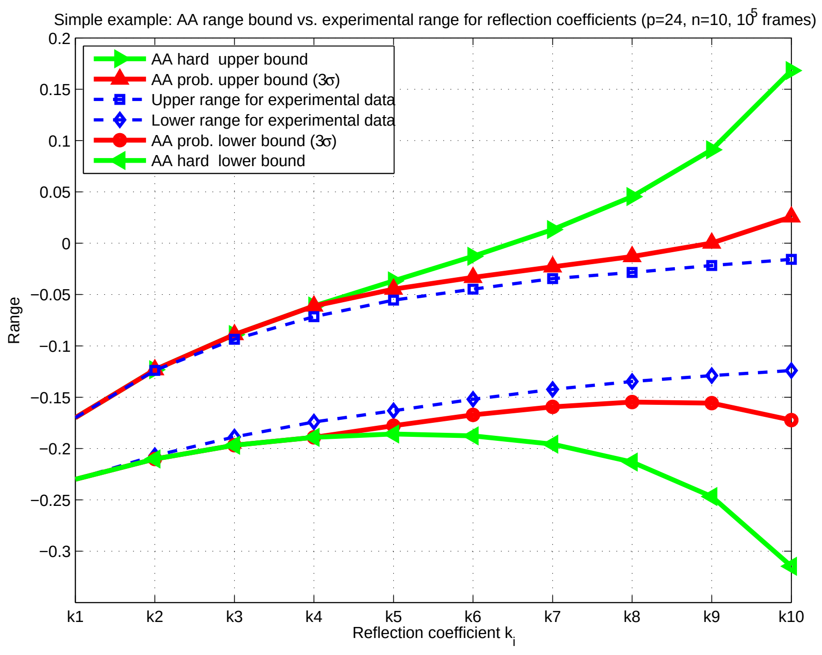


Figure 5.18 The range bound for reflection coefficients k_i for the single-precision Levinson-Durbin algorithm at $n = 10$

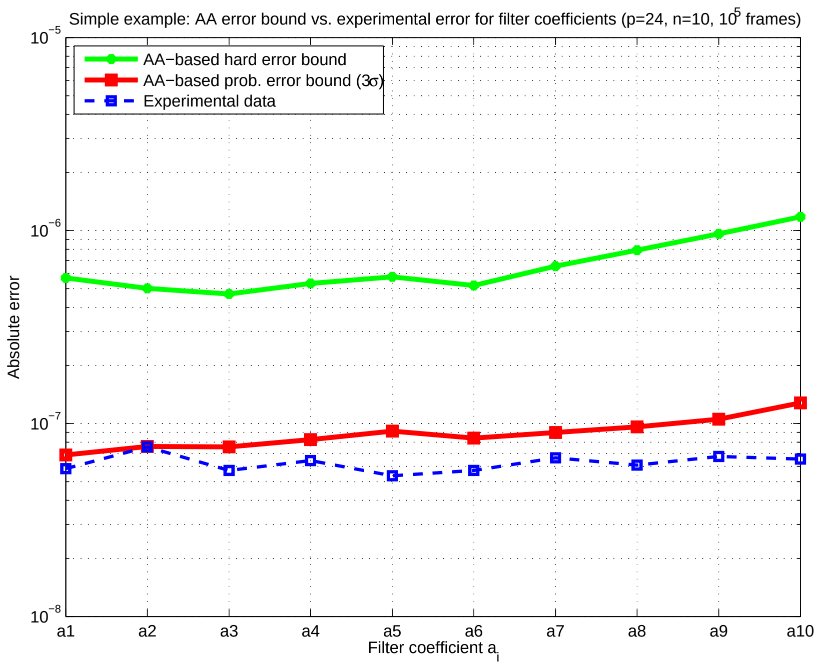


Figure 5.19 The error bound for filter coefficients a_i for the single-precision Levinson-Durbin algorithm at $n = 10$

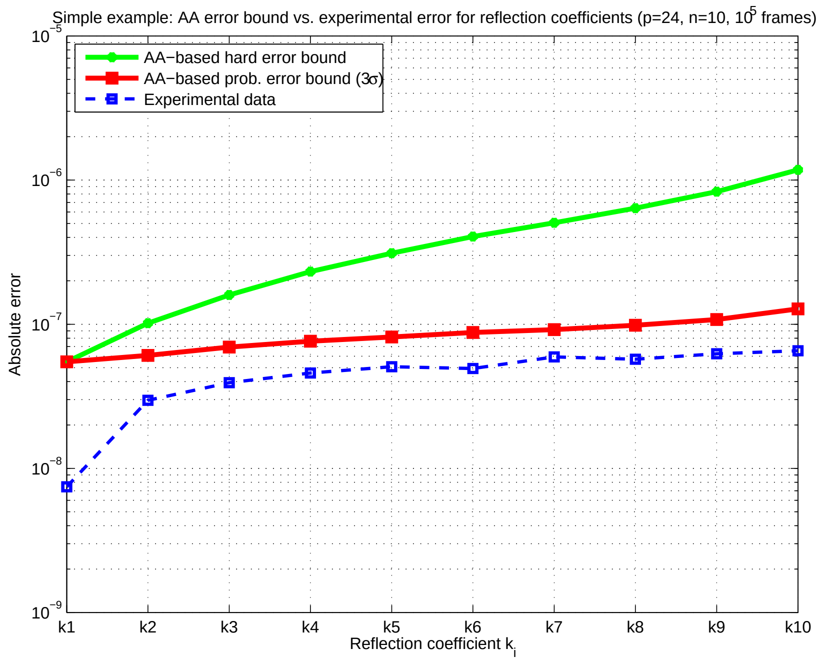


Figure 5.20 The error bound for reflection coefficients k_i for the single-precision Levinson-Durbin algorithm at $n = 10$

Table 5.7 Overestimation ratio for the AA-based range bound of coefficients in the LD algorithm

	Hard bound	Prob. bound (3σ)
Filter coefficient a_i	4.5	2.2
Reflection coefficient k_i	4.5	1.8

range bound for $k_m = -\beta_m/E_{m-1}$ in Figure 5.18, which seems to be due to the decrease in the prediction error E_{m-1} and the increase of β_m over iterations, especially for high indices.

The AA-based rounding error bounds for the filter and reflection coefficients in the LD algorithm are reported in Figures 5.19 and 5.20, respectively. Using the hard bounding operator, the hard error bounds for the filter and reflection coefficients overestimate by an order of magnitude compared to the experimental rounding errors. Using the probabilistic bounding operator, the overestimation ratios of the rounding error bounds for both the filter and reflection coefficients in the LD implementation are about 1.95 times larger than the experimental rounding errors, again, showing that the AA-based probabilistic rounding error bound for the filter and reflection coefficients of the LD algorithm is a reliable estimate.

Case 2: General range for input parameters and using default setting of the `AAFloat` tool

In the second investigation, we first start with the same parameters as used in scenario 1, i.e., $n = 10$, $r_i \in [0.17,\ 0.23]$. We then either increase the order n (as in scenario 2) or specify a general range for the autocorrelation coefficients r_i (scenario 3).

In scenario 2, we keep the same range for the autocorrelation coefficients, i.e., $r_i \in [0.17,\ 0.23]$, and the same working precision, i.e., $p = 24$, while gradually increasing the order n. We use the same Matlab code for the AA-based error analysis of the LD algorithm as used in case 1. The Matlab code is listed in Appendix D.2.

The Matlab program using the `AAFloat` tool for AA-based error analysis of the LD algorithm runs smoothly without any warning or error message up to the iteration $n = 12$. At the iteration $n = 13$, the `AAFloat` tool reported a warning message for *"division by zero"* when performing the inverse operation for the computation of the range of the reflection coefficient k_{13}. It is even worse at the iteration $n = 14$, when the `AAFloat` tool, again, reported warning and error messages for *"division by zero"* in the calculation of the range and error components of the reflection coefficient k_{14}. These observations were consistently reported for both cases: using a hard bounding operator and using a probabilistic bounding operator and at the same iterations, i.e., $n = 13$ and $n = 14$. The full outputs of scenario 2 are reported in Appendix E.1.

More importantly, at the iterations where the division by zero happens, we observe an explosion of the range and error bounds for the filter and reflection coefficients. For example, at $n = 13$, the range and error bound of coefficients in the LD algorithm is reported by the `AAFloat` tool as follows

```
---------- Iteration n = 13
Levinson-Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 1.149381e-05, range = [-0.546389, 2.243222]
error bound of beta    = 1.086233e-05, range = [-1.376382, 1.473215]

Warning: The input range [-0.546389, 2.243222] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1072
  In AA_bound_levinson_wAAFloat_simple_example_increase_order at 88
The user does not specify any range for the input interval of the inverse operation.
AAFloat is using the default range [1.192093e-07, 2.243222e+00] to compute the inverse.
...
i=13, error bound of k(13) = 1.191544e+09, range = [-12358221.218054, 11952076.988843]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.437416e+09, range = [-14850247.879990, 14884126.670198]
i= 2, error bound of a( 2) = 1.111514e+09, range = [-11444673.384395, 11471428.849417]
i= 3, error bound of a( 3) = 1.006093e+09, range = [-10350033.222404, 10375509.506310]
i= 4, error bound of a( 4) = 9.539665e+08, range = [-9809606.568093, 9834651.251699]
i= 5, error bound of a( 5) = 9.261817e+08, range = [-9521868.928906, 9546722.165494]
i= 6, error bound of a( 6) = 8.958615e+08, range = [-9209071.109641, 9233819.194089]
i= 7, error bound of a( 7) = 8.588075e+08, range = [-8823044.851202, 8847725.912109]
i= 8, error bound of a( 8) = 9.029951e+08, range = [-9279778.435664, 9304406.981187]
i= 9, error bound of a( 9) = 9.330091e+08, range = [-9589077.095652, 9613655.846475]
i=10, error bound of a(10) = 9.610570e+08, range = [-9877425.943302, 9901943.523795]
i=11, error bound of a(11) = 1.025096e+09, range = [-10535360.982528, 10559785.600643]
i=12, error bound of a(12) = 1.154501e+09, range = [-11863667.952314, 11887903.841340]
i=13, error bound of a(13) = 1.191544e+09, range = [-12358221.218054, 11952076.988843]

error bound of E_{m} = 3.072350e+09, range = [-27933964.201552, 27914301.953442]
```

This is due to the fact that, in the inverse operation when the input interval contains zero and no input range for handling division by zero is specified, the `AAFloat` class used the default setting $2u$, where u

is the unit roundoff corresponding to the working precision p, for the lower bound of the range of the prediction error E_m. At $p = 24$, the default lower bound is in the order of 10^{-7}, resulting in the huge increases in ranges (of order 10^7) and rounding errors (of order 10^9) of the reflection coefficient k_{13} and all the filter coefficients a_i ($i = 1 \ldots 13$) at that iteration. Furthermore, at the iteration $n = 14$, the `AAFloat` tool gives an error message and stops the program because the lower bound of an interval occurring in the evaluation of the rounding error component of E_m is very close to zero.

The division by zero problem, happening when evaluating the reflection coefficient k_m in the LD algorithm, is more serious in the case of using a general range for the autocorrelation coefficients. In scenario 3, we use the maximum range for the autocorrelation coefficients, i.e., $r_i \in [-1, 1]$, in which each autocorrelation coefficient r_i is represented by an AA form $\hat{r}_i$ as

$$\hat{r}_i = 0 + 1\epsilon_i, \quad \epsilon_i \in [-1, +1], \quad i = 1, 2, \ldots, n,$$

i.e., this AA form has a zero central value and a unit deviation. The full outputs for scenario 3 when using the `AAFloat` class in Matlab are reported in Appendix E.2.

At iteration $n = 2$, the prediction error E_1 has a range of $[0,\ 2]$ (see Appendix E.2), resulting in a division by zero in the evaluation of the reflection coefficient $k_2 = -\beta_2/E_1$. In order to continue the AA-based evaluation, the `AAFloat` tool has to use a default lower bound $2u \approx 10^{-7}$, corresponding to the precision $p = 24$, for the range of E_m. This setting results in a significant increase in the range and error of the filter and reflection coefficients at iteration 2 (i.e., in the order of 10^7). At iteration $n = 3$, the AA-based evaluation for the LD algorithm has to stop because the prediction error E_2, computed at the end of the iteration 2, becomes extremely small.

Case 3: General range for input parameters and using user's setting

In this case, we use a lower bound (in absolute value) for the prediction error E_m which is then used by the `AAFloat` class when the division by zero occurs. This investigation case includes two scenarios, where either the order n is increased while the input range of the autocorrelation coefficients is restricted in the range $[0.17,\ 0.23]$ (scenario 4), or the order n is fixed while a general range of $[-1,\ 1]$ for the autocorrelation coefficients is employed (scenario 5). By specifying the user's range, we hope that better range and error bounds for the quantities in the LD algorithm may be obtainable.

An important issue is identifying the range for the prediction error to be used as the input to the inverse operation in the LD algorithm. As the smallest value of the prediction error reported by simulations with speech signals and synthetic data is on the order of 10^{-4}, we choose the lower bound for the prediction error E_m as $\delta_E = 10^{-4}$. At each iteration, we use the current upper bound for the range of the prediction error E_m as the user-specified upper bound for E_m. The full running reports for scenarios 4 and 5 are listed in Appendices E.3 and E.4, respectively.

By using the user-specified range, the AA-based error evaluation process using the `AAFloat` tool for the Levinson-Durbin iterations can execute up to iteration 14. The evaluation process fails and stops at iteration 15. With respect to the resulting bounds for the range and the rounding error, the use of a user-specified range for the prediction error can provide less overestimated bounds than the use of the default range of the tool.

In scenario 4, by specifying the user's range, the AA-based error evaluation for the LD iterations can execute up to iteration 14. The evaluation process fails and stops at iteration 15. The resulting bounds for the range and rounding error in scenario 4 are less overestimated than the respective bounds in scenario 2. At iteration 13, the range and error bounds reported in scenario 4 for the reflection coefficient k_{13} are approximately 10^4 and 10^3 (see Appendix E.3), respectively, compared to the respective resulting ranges of 10^7 and 10^9 reported in scenario 2 (see Appendix E.1), corresponding to an accuracy improvement of three and six orders of magnitude for the range and rounding error of the reflection coefficient.

Similarly, comparing between scenarios 5 (cf. Appendix E.4) and 3 (cf. Appendix E.2), we observe an accuracy improvement in the range and error bounds for the reflection coefficient k_2 at the second iteration of about three and six orders of magnitude, respectively.

Although the user-specified lower bound for the prediction error is employed, the two scenarios 4 and 5 using the general parameters for the autocorrelation coefficients and system order still fail to execute in all cases and provide very pessimistic bounds compared to the realistic error in practical applications.

Discussion

We have shown that the AA error model with a probabilistic bounding operator can model quite accurately (in the first scenario) the range and rounding error of the LD algorithm with a restricted input data range, i.e., when $r_i \in [0.17,\ 0.23]$, and at the system order $n = 10$. However, in general cases, the division by zero happens when increasing either the system order (as in scenarios 2 and 4) or the range of the input data, i.e., $r_i \in [-1, 1]$ (as in scenarios 3 and 5). The division by zero occurs because the input interval of the prediction error E_m contains zero, reported by the `AAFloat` class. Here we discuss the AA-based evaluation of the prediction error E_m in the LD algorithm and the meaning of the prediction error E_m with respect to the numerical properties of the LD algorithm.

In linear prediction, the size of the prediction error E_m is critical to the stability and the numerical properties of the LD algorithm [30]. The Toeplitz system is stable if all the reflection coefficients are smaller than 1 in absolute value. There exists a close relation among the prediction error E_m, the condition number of the Toeplitz system (i.e., the condition number of the autocorrelation matrix) and the reflection coefficients, presented by Cybenko in [30], such that the condition number is guaranteed large if E_m is small. The prediction error is known to be in the range $0 \leq E_m \leq 1$. "E_m is small" means it is very close to zero. In turn, the prediction error E_m is small if any of the reflection coefficient is close to 1 in absolute value, in other words, the Toeplitz system comes close to an unstable system.

However, in AA-based error analysis, the range for the prediction error E_m is not guaranteed to be in $[0,\ 1]$ but it increases with the system order. This can be explained as follows. At the m-th iteration, the prediction error E_m can alternatively be computed via the autocorrelation coefficients and the filter coefficients as

$$E_m = \sum_{i=0}^{m} a_i^{(m)} r_i.$$

In AA-based error analysis, as each autocorrelation coefficient r_i is distributed over the symmetric range $[-1, 1]$ and each filter coefficient a_i follows a binomial function (see 5.3.2), the numerical range for E_m can increase up to $[-2^m,\ +2^m]$ at the m-th iteration, i.e., E_m contains zero. As a consequence, the inverse operation of E_m for computing the reflection coefficient k_{m+1} at the next iteration may result in a division by zero (as observed in scenarios 3 and 5). In the case of using a restricted range $[0.17,\ 0.23]$ for the coefficient r_i, the range of the prediction error E_m may still have the possibility to reach zero due to the increase of the system order n and the overestimation effect of non-affine operations in the LD iterations. This observation was reported in scenarios 2 and 4.

In fact, in the first scenario using $r_i \in [0.17,\ 0.23]$ and a system order 10, we observed that all the condition numbers of all the Toeplitz systems, used in simulations and corresponding to the chosen range, are smaller than or equal to 6.4 and the resulting prediction error is in the range $0.62 \leq E_m \leq 1,\ m = 1 \ldots 10$, or all the chosen Toeplitz systems are very well-conditioned. With respect to AA-based error analysis, the AA-based range bound for the prediction error at iteration $m = 10$ is $[0.63,\ 1.08]$, reported by the `AAFloat` tool. When increasing the system order (i.e., in the second investigation), the AA-based range bound for E_m increases to $[0.42,\ 1.29]$ and $[-0.54,\ 1.29]$ at $m = 11$ and $m = 12$, respectively, and rapidly rises to approximately $[-2.8 \times 10^7,\ 2.8 \times 10^7]$ at $m = 13$.

With respect to the term β_m in the LD algorithm, i.e., equation (5.12), it is easy to show for a well-conditioned system in linear prediction that β_m is bounded in the range $[-1, 1]$ and smaller than the prediction error E_{m-1} in absolute value such that the computed reflection coefficient $k_m = -\beta_m / E_{m-1}$ is always guaranteed to be in the range $[-1, 1]$. This is due to the fact that the true values of autocorrelation coefficients r_i and filter coefficients a_i in real-world applications can be either positive or negative, such that cancellation in the computation of β_m will guarantee $-1 \leq \beta_m \leq 1$. For AA-based error analysis, this property is no longer guaranteed. The range bound for β_m can therefore go up to $[-2^{m-1},\ +2^{m-1}]$ at the m-th iteration (provided that there is no overestimation for β_m), which may however still result in the reflection coefficient bounded in the range $[-1, 1]$.

Our analysis above shows the following limitations of the current AA-based error model for error analysis of the LD algorithm in a general case of having maximum input data range $[-1, 1]$ and an arbitrarily large system order:

- ❏ First, the ranges estimated by the AA-based model for the terms β_m and E_m in the LD algorithm cannot accurately capture the theoretical range given by analytical work in the literature [7, 30].
- ❏ Second, as the range of E_m increases along with the system order and can contain zero value, division by zero is unavoidable when computing the reflection coefficients with the AA model in the LD algorithm, making the AA-based model only applicable for the special case of having both a restricted input data range and a small system order.
- ❏ Even though the `AAFloat` tool allows users to specify a lower bound for the input interval of the inverse operation to alleviate somewhat the effect of division by zero, the division operation in the LD algorithm can certainly cause very large errors if the chosen prediction error E_m is very close to zero, which is equivalent to having a high condition number for the autocorrelation matrix.
- ❏ Furthermore, the overestimation effects due to a long computational chain of non-affine operations may give rise to pessimistic range and error bounds for the quantities in the LD algorithm.

It is not easy to overcome all the limitations listed above. We are aware of related work in [4] using an asymmetric probabilistic bounding operator to improve the accuracy of non-affine operations. However, our goal is to use a new Matlab tool for efficiently performing error analysis of floating-point algorithms using the ordinary AA model and to investigate how well the AA model and the `AAFloat` tool is applicable for real-world applications. Therefore, we will stay with the ordinary AA-based error model and accept the overestimation effect of non-affine operations as part of the model.

Recall that the autocorrelation sequences as the input to the LD algorithm must be generated in a proper way, meaning that the arbitrary choice of a sequence $\{r_0, r_1, \ldots, r_n\}$ with $r_i \in [-1,\ 1]$ will

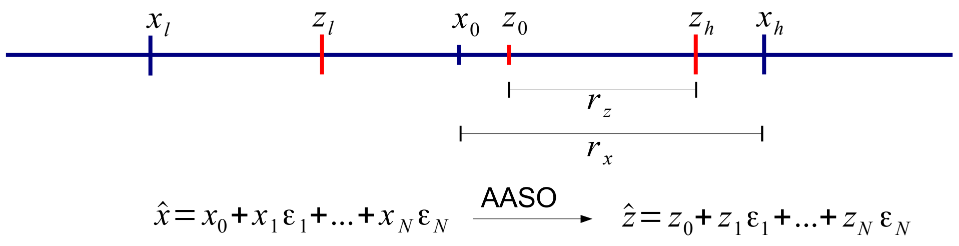

x_l
z_l
x_0
z_0
z_h
x_h
r_z
r_x
$\hat{x}=x_0+x_1\varepsilon_1+...+x_N\varepsilon_N$
AASO
$\hat{z}=z_0+z_1\varepsilon_1+...+z_N\varepsilon_N$

Figure 5.21 Basic principle of the AA-based scaling operator (AASO)

not result in a valid autocorrelation sequence and, therefore, cannot correspond to any (stable) linear prediction filter (even at the infinite precision number format). Therefore, for AA-based error analysis, it would be more realistic to set up a data generator for the autocorrelation coefficients r_i from the reflection coefficients k_i, all of these things should be implemented in AA expressions using the `AAFloat` class, and then use the estimated coefficients r_i as the input to the AA-based error analysis of the LD algorithm, again, using the `AAFloat` class. By doing like that, the stability of the LD algorithm can still be guaranteed (i.e., $|k_m| \leq 1$) and we hope that, by using a more realistic data distribution for r_i, the correlation among the autocorrelation coefficients r_i may be captured accurately, therefore it might hopefully result in more precise range and rounding error bounds for quantities in the LD algorithm. However, that setting is quite complex and potentially contains sources of overestimation in the generation of r_i from k_i due to non-affine operations, and therefore it goes far beyond the scope of this thesis.

In fact, the LD algorithm has been well studied in the literature [7, 8, 30], allowing us to have knowledge of the ranges of all the quantities in the LD algorithm without having to run any AA-based range evaluation. Hence we only need to estimate the rounding error of the LD algorithm using the `AAFloat` class. The remaining challenge is, however, to handle division by zero in the LD algorithm. Note that the division by zero is not a limitation of the AA model or the `AAFloat` tool, it is a problem of the LD algorithm itself. This task is of our interest.

In the next subsection, we suggest to use an AA scaling operation to enforce the range of the quantities E_m, β_m and k_m on their known ranges provided by analytical studies in literature, then estimate the rounding error based on the enforced range.

AA-based scaling operator

We introduce an AA-based scaling operator (AASO) for converting an AA form $\hat{x}$ representing an interval $[x_l,\ x_h]$ to another AA form $\hat{z}$ representing the interval $[z_l,\ z_h]$ such that $x_l \leq z_l < z_h \leq x_h$, i.e., the scaled AA form $\hat{z}$ implies a smaller range than the one implied by the original AA form $\hat{x}$. The basic idea of the AASO is descibed by Figure 5.21. In this figure, we assume

$$\hat{x} = x_0 + x_1\epsilon_1 + \cdots + x_N\epsilon_N$$

is the original AA form that represents the original interval $[x_l,\ x_h] = [x_0 - r_x,\ x_0 + r_x]$ where r_x is the total deviation of $\hat{x}$. We want to have a new AA form

$$\hat{z} = z_0 + z_1\epsilon_1 + \cdots + z_N\epsilon_N$$

representing the restricted interval $[z_l,\ z_h]$ specified/required by users. The two AA forms have the same number of noise terms. Given the AA form $\hat{x}$ and the interval $[z_l,\ z_h]$ (i.e., assuming $z_l < z_h$) as input parameters to the AASO, the new AA form $\hat{z}$ is constructed by the following algorithm:

$$\begin{aligned}
r_x &= \sum_{i=1}^{N} |x_i| \\
x_l &= x_0 - r_x \\
x_h &= x_0 + r_x \\
z_l &= \max\{z_l,\ x_l\} \\
z_h &= \min\{z_h,\ x_h\} \\
z_0 &= (z_h + z_l)/2 \\
r_z &= (z_h - z_l)/2 \\
\theta &= r_z/r_x, \qquad\qquad (0 < \theta \leq 1) \\
\hat{z} &= z_0 + \sum_{i=1}^{N} z_i \epsilon_i, \quad \text{with } z_i = \theta x_i \ (i = 1 \ldots N),
\end{aligned}$$

where the `max` and `min` operations are employed to make sure that $x_l \leq z_l < z_h \leq x_h$. The scaling constant θ is in the range $(0, 1]$.

The AASO is similar to the affine scaling operation, i.e., multiplication with a constant (cf. subsection 3.2.3), but not identical. This is because the AASO performs two operations: shifting the central value from x_0 to z_0, and scaling the partial deviation x_i to z_i by the factor θ. Shifting the central value is necessary for preserving the symmetry property of the new affine form $\hat{z}$. The AASO applied on the affine form $\hat{x}$ is, therefore, equivalent to *i)* adding $\hat{x}$ with a constant c, and *ii)* multiplying with a scaling factor θ as follows

$$\hat{z} = \texttt{AASO}(\hat{x}) = (\hat{x} + c)\theta,$$

where

$$c = \frac{z_0}{\theta} - x_0, \qquad 0 < \theta \leq 1.$$

If $\theta = 1$ then $c = 0$ and $z_i = x_i$ $(i = 0, 1, \ldots, N)$, i.e., there is neither central value shift nor partial deviation scaling or $\hat{z} = \hat{x}$. Since the AASO involves performing affine operations only, there is no overestimation due to the AASO.

How can we use the AA-based scaling operator described by the algorithm above to enforce the range of an AA expression $\hat{x}_f = \hat{x}_f^r + \hat{x}_f^e$, representing a floating-point variable x_f, in order to have a new AA expression $\hat{z}_f = \hat{z}_f^r + \hat{z}_f^e$? In the `AAFloat` tool, the range component $\hat{z}_f^r$ is computed by applying exactly the algorithm above. Since the rounding error of a floating-point number is directly proportional to the range of the number, the error bound is expectedly scaled by a factor of θ, approximately. The error component is evaluated by scaling with the scaling constant θ only: $\hat{z}_f^e = \theta \cdot \hat{z}_f^e$, as a central value is not used for the error component. In the `AAFloat` tool, the AASO is implemented by the method `Z_AA = AASO(X_AA,userrange)` where the input parameter `userrange` specifies the new range $[z_l,\ z_h]$.

AA-based error analysis for Levinson-Durbin algorithm with an AA-based scaling operator

We employ the AA-based range scaling operator to perform floating-point error analysis of the LD algorithm in a general case. The basic idea is to incorporate the analytical knowledge on the ranges of all quantities in the LD algorithm, given in literature, into the AA-based evaluation of the algorithm. In other words, we only focus on rounding error estimation. At each iteration, the AA-based scaling operator is used to enforce the known ranges of the terms β_m, E_m, k_m and a_i. In fact, at iteration m, we observe that only the range bounds for the term β_m, the prediction error E_m and the reflection coefficient k_m need to be enforced. The range for the filter coefficient a_i is automatically enforced due to the computational dependencies within the LD algorithm. The enforced range bounds for the terms β_m, E_m and k_m are

$$\begin{aligned} -1 \leq \beta_m \leq 1, \\ -1 \leq k_m \leq 1, \\ \delta_E \leq E_m \leq 1, \end{aligned}$$

where we choose a value $\delta_E = 10^{-4}$ from experimental simulations as the lower bound of the prediction error E_m. This is done by adding three calls to the method AASO of the `AAFloat` class right after evaluating the respective terms, as shown by the following segment of code:

```
delta_E = 0.0001;
...
LD_beta = mmul(r, (fliplr(a))', prob);     % Step 1: compute the numerator beta
LD_beta = AASO(LD_beta, [-1, 1]);          % 1a: enforce LD_beta into the range [-1, 1]
...
kk = -div(LD_beta, LD_alpha, prob);        % Step 2: compute reflection coefficient k
kk = AASO(kk, [-1, 1]);                    % 2a: enforce k into the range [-1, 1]
...
kk2         = sqr(kk, prob);               % Step 4: compute prediction error E
kk2_1       = sub(1, kk2, prob);           %
LD_alpha    = mul(LD_alpha, kk2_1, prob);  % E = E(1-k*k)
LD_alpha = AASO (LD_alpha, [delta_E, 1]);  % 4a: enforce LD_alpha into the range [delta_E, 1]
...
```

The full Matlab code used with the `AAFloat` tool is listed in Appendix D.3. The input range for the autocorrelation coefficients r_i is set to $[-1, 1]$. The system order is $n = 10$. Both the hard and probabilistic error bounds are investigated.

The AA-based error bound evaluation of the LD algorithm using the AASO is performed successfully without any warning or error messages until the last iteration ($n = 10$).[5] The full reports in Matlab are listed in Appendix E.5 and E.6 for the hard and probabilistic error bounds, respectively. We observed that all the numerical range bounds for all the quantities at each iteration are in a reasonable range compared to the analytical results from the literature. We make a simple comparison on the *hard* error bound of the *reflection coefficient* k_2 at iteration 2 in two cases: (a) not using an AA-based range scaling operator in scenario 5 (cf. Appendix E.4) and (b) using an AA-based range scaling operator (cf. Appendix E.5). The latter gives a much smaller error bound for the reflection coefficient k_2 than the former, i.e., 6.56×10^{-3} compared to $8.35 \times 10^{+1}$ or approximately four orders of magnitude. This is obviously due to the use of the AA-based range scaling operation.

Now we take a closer look at the error bounds for the LD algorithm at order $n = 10$. Figures 5.22 and 5.23 plot the resulting error bounds for the filter and reflection coefficients when the AA-based

[5]In fact, we also tried with a system order $n = 100$ and the AA-based error evaluation was executed properly with the `AAFloat` tool up to the iteration 89. From iteration 90, the computed bounds exceed the representable range of floating-point numbers in Matlab, and the `AAFloat` tool reports the real range $\mathbf{R}$ (∞). For the sake of simplicity, the numerical results corresponding to the order 100 will not be presented here.

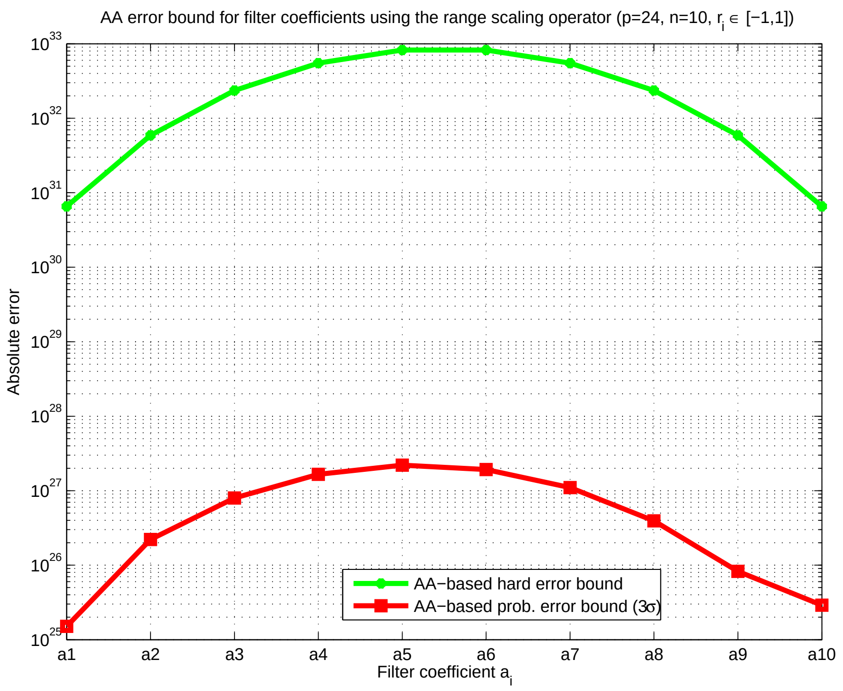


Figure 5.22 AA error bound for filter coefficients when using an AA-based scaling operator

scaling operator is applied to enforce the ranges of the terms in the LD iterations. For filter coefficients, the hard error bound is about four orders of magnitude larger than the probabilistic error bound. For reflection coefficients, the hard error bound is slightly larger than the probabilistic one. In comparison with the experimental rounding errors for speech signal and synthetic data, which are reported by Figures 5.12 and 5.13 in Section 5.3.4, the AA-based error bounds for the filter coefficients and the reflection coefficients of the LD algorithm are extremely pessimistic.

We would like to understand why the error bounds for the filter and reflection coefficients are so pessimistic. We study in detail the AA-based error evaluation for the reflection coefficients using the AASO. At iteration m, the reflection coefficient k_m is computed as $k_m = -\beta_m / E_{m-1}$. Since the ranges of the terms β_m and E_{m-1} are enforced by the AASO before computing the reflection coefficients, the

Table 5.8 Hard error bound for the reflection coefficient k_m over the Levinson-Durbin iterations ($p = 24$)

Iteration	Scaled error bound of E_{m-1}	Error bound of k_m before performing AASO	Error bound of k_m after performing AASO
$m = 1$	0	2.4×10^{-7}	2.4×10^{-7}
$m = 2$	6.6×10^{-7}	$6.6 \times 10^{+1}$	6.6×10^{-3}
$m = 3$	8.7×10^{-3}	$8.7 \times 10^{+5}$	$8.7 \times 10^{+1}$
$m = 4$	$1.2 \times 10^{+2}$	$1.2 \times 10^{+10}$	$1.2 \times 10^{+6}$

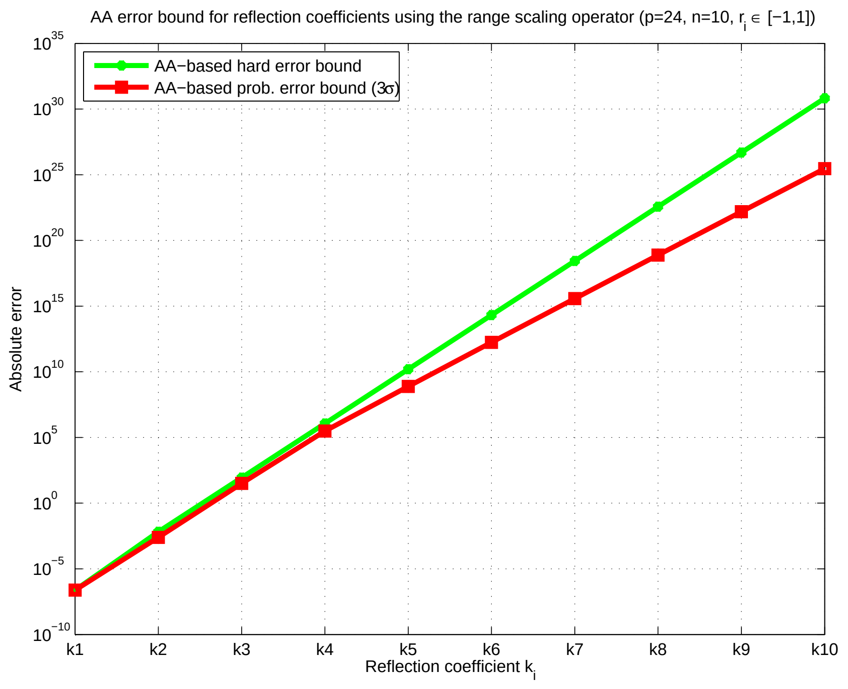


Figure 5.23 AA error bound for reflection coefficients when using an AA-based scaling operator

range bound for k_m can be estimated as follows

$$[k_m] = -\frac{[\beta_m]}{[E_{m-1}]} = -\frac{[-1,1]}{[\delta_E, 1]} \approx [-\frac{1}{\delta_E},\ \frac{1}{\delta_E}],$$

where $[\cdot]$ is for representing the range. It is clearly shown that the range bound for k_m depends on the lower bound δ_E of the prediction error. With the chosen $\delta_E = 10^{-4}$, the unscaled range bound for k_m can be up to $[-10^4,\ 10^4]$, which is then reduced by the AASO to the range $[-1, 1]$ with a scaling factor of $1/\delta_E$.

According to the evaluation of the error bound for the inverse/reciprocal operation in equation (3.30), the error bound for k_m before performing the AASO can be approximated as the error bound of E_{m-1} multiplied by $(1/\delta_E)^2$. After performing the AASO, the error bound of k_m is reduced by a factor of $1/\delta_E$. With $\delta_E = 10^{-4}$, $(1/\delta_E)^2 = 10^8$ and $1/\delta_E = 10^4$. Table 5.8, extracted from Appendix E.5, reports the hard error bound for the reflection coefficient k_m, before and after performing the AASO, over some iterations of the LD algorithm at single-precision format ($p = 24$). On average, the AA-based hard error bound for the reflection coefficient increases about three or four orders of magnitude after each iteration.

The analysis presented here gives a reasonable explanation for the pessimistic estimate for the error bound of the LD algorithm using the AA-based error model. We observe that the size of the prediction error is crucial to the overall rounding errors of the reflection and filter coefficients. It seems that the division operation is the main source of error in the LD algorithm as it allows for the possibility of significant rounding error magnification.

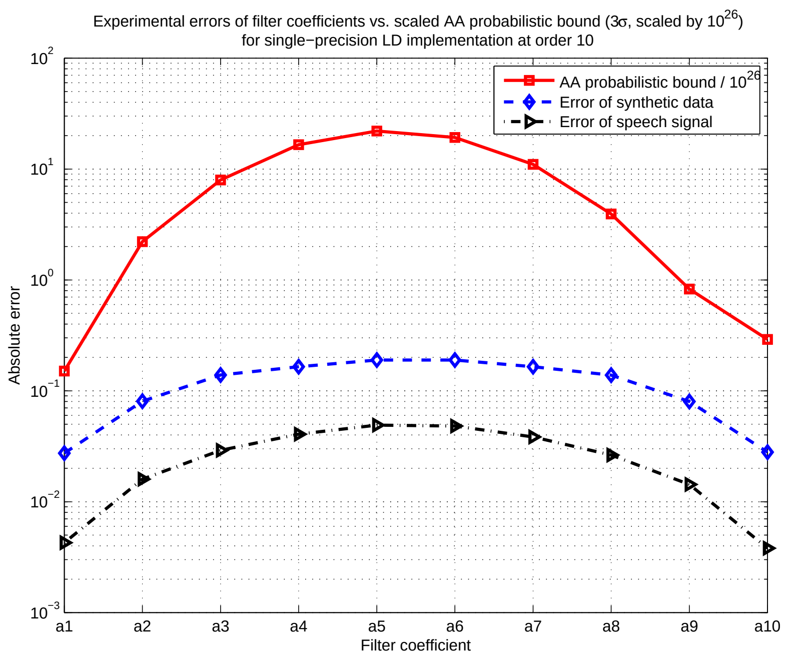


Figure 5.24 Experimental rounding errors ($\kappa \leq 10^6$) of filter coefficients vs. scaled AA-based probabilistic error bound (3σ, scaled by 10^{26}) for single-precision Levinson-Durbin implementation at order 10

AA-based error bound versus experimental error for filter coefficients. To finalise the AA-based error analysis using the AA-based scaling operator, we compare the error bounds estimated using the scaling operator with the experimental rounding error for the filter coefficients. The comparison is shown in Figure 5.24. The less pessimistic error bound, i.e., the AA-based probabilistic bound (with a confidence interval 3σ), is chosen for comparison. Similarly, the maximum experimental rounding errors, corresponding to the highest condition number of $\kappa \leq 10^6$, for the synthetic data and speech signals are chosen (i.e., the two error curves shown in Figure 5.24 are identical to the two highest error curves presented earlier in Figure 5.12 for the synthetic data and speech signals, respectively).

Since the AA error bound is overestimated compared to the experimental error, the AA error bound for the filter coefficient is scaled by a factor of 10^{26} for a better representation in the same figure (i.e., the true error bound equals the represented error bound in Figure 5.24 multiplied by 10^{26}) and we only focus on the trend of the error bound. The trend of the AA error bound almost matches the trend of the experimental errors for filter coefficient a_i, clearly showing that the AA-based model with a scaling range operator is able to provide a fairly good qualitative estimate for the rounding error of the filter coefficients a_i in the iterative LD algorithm.

5.3.6 Summary

In the second application of AA-based floating-point error analysis, we studied the rounding error of the iterative LD algorithm with respect to the system order, the precision and the condition number. The size of the prediction error E_{m-1} and the division have a major impact on the accuracy of the

LD algorithm. To what extent these two factors affect the estimated bound depends on the respective input data, for which different scenarios have been studied.

The AA error model with a probabilistic bounding operator can model accurately the range and rounding error of the LD algorithm when using restricted input parameters, i.e., $r_i \in [0.17,\ 0.23]$ and an order $n = 10$, in which the corresponding overestimation ratio for the error of the filter coefficients and the reflection coefficients is approximately 1.95. For bit width allocation, this is equivalent to an overestimation of 1 mantissa bit.

In case of using general parameters for the AA error evaluation of the LD algorithm, i.e., $r_i \in [-1, 1]$ and arbitrarily large order n, the most challenging issue is handling the division by zero happening due to the severe overestimation of the prediction error over iterations of the algorithm. In linear prediction, division by zero in the LD iterations may happen if the prediction error E_m decreases very close to zero, which is equivalent to the fact that the Toeplitz system at hand has very high condition number or it is ill-conditioned. The `AAFloat` tool allows us to specify a lower bound for the prediction error, which, however, cannot help overcome the division by zero problem. We then suggest to use the AA-based range scaling operator to reduce the range bounds of the quantities in the LD algorithm to their known ranges given by theoretical studies in the literature. The AA-based probabilistic error model and the range scaling operator can provide a good qualitative estimate for the error bound of the LD algorithm. The resulting error bounds for the filter coefficients and reflection coefficients are, however, very pessimistic compared to the real errors observed in practical applications, making these bounds useless for bit width allocation for hardware implementation of the LD algorithm.

Another important topic is performing the custom-precision floating-point LD implementation. The experimental setup presents different scenarios for rounding error evaluation of the custom-precision LD implementation. Users should be aware of the numerical stability of the algorithm and, therefore, they need to make sure that the reflection coefficients are always in the range $[-1, 1]$ for guaranteeing the stability of the algorithm when setting up the experiments as well as during the execution of the LD implementation.

Experimental results show that the filter coefficients a_i are more sensitive to the rounding error than the reflection coefficients k_i. The rounding error is directly proportional to the condition number of the Toeplitz matrix. We observe that more than half of the speech frames correspond to ill-conditioned Toeplitz systems having condition numbers of 10^4 or higher. The average spectral distortion for speech suggests to use a 16 mantissa bits number format for the floating-point implementation of LPC vocoders.

5.4 Conclusions

The chapter presented two examples of AA-based floating-point rounding error estimation for two signal and speech processing applications: dot-product and linear prediction with the Levinson-Durbin algorithm.

It has been shown in the first example - error analysis of the floating-point dot-product - that the AA-based probabilistic rounding operator can provide tighter estimated rounding error bounds in comparison with the conventional error modeling by Higham [5]. The AA-based probabilistic rounding error analysis is, therefore, a promising technique for error analysis of floating-point algorithms consisting of affine operations like the dot-product, which is very useful for the estimation of optimal uniform bit width for floating-point implementations of respective applications on reconfigurable hardware.

For the case of a complex algorithm consisting of multiple non-affine operations, especially when an algorithm needs to compute a division/inverse, as in the case of the Levinson-Durbin algorithm, the AA model may result in very pessimistic error bound due to the overestimation effect of non-affine operations and/or it may even fail to execute the evaluation due to division by zero. If there exists theoretical analysis of the algorithm at hand, one can try to incorporate the additional knowledge on the algorithm from analytical work with the AA model in order to somehow alleviate the overestimation effect and obtain a more sensible error estimate.

In general, the faster execution time offered by the AA-based model and the `AAFloat` tool (than simulation approaches) will be a true asset to many practical applications in which the optimal bit width configuration is a demanding input parameter. Potential applications are: *i)* in floating-point code generators, e.g., the FloPoCo project [76, 77] or the SPIRAL project [78–80] specifically aiming at the software/hardware generation for DSP algorithms on FPGAs, or *ii)* in real-time applications with data-dependent bit width optimization, e.g., in a wireless sensor network where the states of nodes are changing over time and the optimal resource usage (with respect to energy, memory, communication bandwidth) is demanding and obtainable with custom-precision computations.

6 Conclusion

This chapter gives a summary of the scientific contributions of this doctoral thesis and discusses potential future research directions.

6.1 Scientific Contributions

This doctoral work aims at efficient floating-point implementation of signal processing algorithms on reconfigurable hardware by exploiting custom-precision floating-point operations and performing floating-point rounding error analysis and optimal uniform bit width allocation. The scientific contributions of this work include:

Performance of custom-precision floating-point arithmetic operations: A case study on a hybrid reconfigurable CPU

We investigate the area performance and throughput performance of custom-precision floating-point arithmetic operations via the implementation of a floating-point fused multiply-accumulate operation on the Stretch S6 prototypical hybrid reconfigurable CPU.

It is known that reduced precision in floating-point arithmetic operations on reconfigurable fabrics directly translates into increased parallelism and peak performance, thereby allowing for trading accuracy with parallelism and performance. We show that the reconfigurable fabric of the S6 CPU is able to provide native support of custom-precision floating-point arithmetic up to double-precision; for single-precision multiple operators can be implemented in parallel. Our investigation on the S6 CPU can be seen as a case study that provides one more piece of evidence for the statement of the tradeoffs in accuracy, parallelism and performance on reconfigurable platforms.

The dominant issue identified while investigating the S6 CPU is the mismatch between the reconfigurable fabric and the I/O bandwidth making hybrid reconfigurable CPUs temporarily unsuitable for scientific workloads.

A Matlab-based framework for floating-point rounding error analysis using affine arithmetic and uniform bit width allocation

We implement the first Matlab-based framework for performing rounding error analysis and numerical range evaluation of arbitrary floating-point algorithms using affine arithmetic error modeling. With the support of basic vector and matrix computations in Matlab, our framework enables users to best reuse their own existing Matlab codes to effectively perform the rounding error analysis task. The `AAFloat` tool supports flexible handling of exceptional cases via providing users useful information and allowing users to specify reasonable ranges in handling the division by zero and square root of negative intervals.

We also incorporate arbitrary-precision floating arithmetic via the GNU MPFR library into the framework, thereby allowing for an efficient bit-true custom-precision computation of basic floating-point arithmetic operations in Matlab.

Besides, we suggest the first AA-based error model for a floating-point fused multiply-accumulate operation $z = xy + w$ that helps to conduct floating-point error analysis of applications involving this fused operation.

The rounding error bound of the floating-point algorithm evaluated by our Matlab-based software tool can be used for optimal uniform bit width allocation. The framework supports both an AA-based hard-bounding operator and an AA-based probabilistic-bounding operator with the confidence interval giving the users more freedom in bit width allocation for floating-point implementation (i.e., trading accuracy for performance and area by choosing between the hard bounding operator and the probabilistic bounding operator in combination with a varying confidence interval).

Rounding error analysis and bit width allocation for floating-point dot-products

We use the Matlab-based framework and the AA-based probabilistic bounding operator to estimate the rounding error bounds of different floating-point dot-product architectures (i.e., using basic operations versus using fused operations, and using sequential versus parallel structures) over a wide range of input parameters. We show that an AA-based probabilistic bounding operator is able to provide a tighter rounding error bound compared to conventional forward error analysis, thereby allowing for minimum mantissa bit width allocation and comparison of different dot-product architectures.

Different dot-product architectures result in different rounding errors. We show that the overall numerical accuracy of floating-point dot-products can considerably be improved by changing from a sequential structure to a parallel structure (at the cost of more hardware resources) but not by changing from using basic operations to using a fused operation. In terms of bit width allocation, we show that the AA-based technique overestimates the required bit width by at most 2 mantissa bits for the floating-point dot-product implementation.

More importantly, we derive in this work the analytical rounding error models for all floating-point dot-product architectures as a function of numerical range, precision, vector length and chosen confidence interval, allowing for an efficient design space exploration and which are key to floating-point code generators.

Rounding error analysis for the floating-point Levinson-Durbin algorithm

The last and very important contribution of this work is the AA-based rounding error analysis for the floating-point Levinson-Durbin algorithm.

We are the first to apply affine arithmetic for the evaluation of the rounding error propagation in the iterative Levinson-Durbin algorithm. This work studies the rounding error of the filter coefficients and reflection coefficients taking into consideration the system order, the precision and the condition number of the Toeplitz matrix. We show that the division for the prediction error in the Levinson-Durbin algorithm significantly affect the resulting rounding errors. We show that for the case of a complex algorithm consisting of multiple non-affine operations, as in the case of the Levinson-Durbin algorithm, the AA model may result in very pessimistic error bound due to the overestimation effect of non-affine operations and/or it may even fail to execute the evaluation due to division by zero. We suggest to incorporate the additional knowledge on the algorithm from analytical work with the AA model, via applying an enforced bound on the range with the AA-based scaling range operator, in order to alleviate the overestimation effect, thereby obtaining more sensible qualitative estimates for the error bounds.

6.2 Future Work

We hope that our work will encourage others to apply affine arithmetic and reconfigurable hardware in the areas of signal/speech processing and scientific computing. Possible directions for future work include:

- **Applying the new method and tools for existing algorithms**. The AA-based error model and Matlab-based tool in this work can equally be applied to arbitrary floating-point algorithms to understand better the usability of the affine arithmetic model and the implemented Matlab-based tool. Potential classes of algorithms for investigation with the new method include recursive algorithms and feedback systems.
- **Improving the accuracy of AA-based (rounding) error models for non-affine operations**. The inherently symmetric property of affine intervals causes very pessimistic range estimates for non-affine operations like multiplication and division. The effect of the symmetry property of affine intervals becomes more serious in computation chains consisting of many non-affine operations. The AA-based range scaling operator suggested in this work still results in pessimistic estimates. Fang [4] suggested to use an asymmetric bounding operator for evaluating non-affine operations, which requires significant computational effort. Therefore, it is desirable to have alternative improved technique based on affine arithmetic that can provide an accurate estimate for non-affine operations and only requires a little more computational effort compared to the current AA-based error model used in this thesis.
- **Performing multiple-precision error analysis and bit width allocation for floating-point algorithms**. The basic idea is that different computational parts of a floating-point algorithm may use different floating-point number formats (different precisions) provided that all computational parts can still guarantee the overall desired accuracy.
- **Combined CPUs and FPGA-based architectures for speech processing**. Exploiting the combination between traditional CPUs and reconfigurable hardware like FPGAs can bring advantages of the two platforms together. The challenge is how to combine best these two technologies at different design levels and what speech processing applications would benefit from using this combination.

A
AA-Based Error Model for Floating-Point Reciprocal

Given the AA expression for a floating-point variable y_f as

$$\begin{aligned}\hat{y}_f &= \hat{y}_f^r + \hat{y}_f^e \\ &= (y_0 + \sum_{i=1}^{N} y_i \epsilon_i) + B(\hat{y}_f^r) \cdot u \cdot \epsilon_y,\end{aligned}$$

we would like to evaluate the reciprocal of this variable: $\hat{z}_f = 1/\hat{y}_f$ such that the range component and rounding error component of $\hat{z}_f$ can be represented via the range component and rounding error component of $\hat{y}_f$ as

$$\hat{z}_f = \frac{1}{\hat{y}_f^r + \hat{y}_f^e} = \hat{z}_f^r + \hat{z}_f^e.$$

We rewrite the above expression as follows:

$$\begin{aligned}\hat{z}_f &= \frac{1}{\hat{y}_f^r + \hat{y}_f^e} \\ &= \frac{1}{\hat{y}_f^r} \cdot \left(1 + \frac{\hat{y}_f^e}{\hat{y}_f^r}\right)^{-1}\end{aligned}$$

Generally, the error component is very much smaller than the range component, i.e., $\hat{y}_f^e \ll \hat{y}_f^r$, we can approximate the second term in the equation above as follows [4]

$$\left(1 + \frac{\hat{y}_f^e}{\hat{y}_f^r}\right)^{-1} \approx 1 - \frac{\hat{y}_f^e}{\hat{y}_f^r},$$

and the reciprocal can therefore be approximated as

$$\hat{z}_f \approx \frac{1}{\hat{y}_f^r} \cdot \left(1 - \frac{\hat{y}_f^e}{\hat{y}_f^r}\right),$$

or

$$\hat{z}_f = \frac{1}{\hat{y}_f^r} - \frac{1}{(\hat{y}_f^r)^2} \cdot \hat{y}_f^e.$$

From this, the range component and the rounding error component of $\hat{z}_f = 1/\hat{y}_f$ can be described as follows:

$$\begin{aligned}
\hat{z}_f^r &= \frac{1}{\hat{y}_f^r}, \\
\hat{z}_f^e &= -\frac{1}{(\hat{y}_f^r)^2} \cdot \hat{y}_f^e + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z,
\end{aligned}$$

where a new error term $B(\hat{z}_f^r) \cdot u \cdot \epsilon_z$ associated with a new noise term ϵ_z is introduced in the rounding error component to represent the rounding error of the reciprocal itself. The remaining error term in the rounding error component $\hat{z}_f^e$ comes from the input operand y, i.e., $\hat{y}_f^e$.

The expressions for the range and error components of $\hat{z}_f$ shown above are not affine forms. In order to have the affine representations for the range and error components of $\hat{z}_f$, we use the *min-range approximation* [53] for the reciprocal as well as apply the bounding operator $B(\cdot)$. The evaluation of all the coefficients of the min-range approximation is presented in Section 3.3.2.

$$\begin{aligned}
\hat{z}_f^r &= f^{\text{min-range}}\left(1/\hat{y}_f^r\right) \\
&= \alpha \cdot \hat{y}_f^r + C_0 + C_1 \cdot \epsilon_{N+1} \\
&= (\alpha \cdot y_0 + C_0) + \sum_{i=1}^{N} y_i \epsilon_i + C_1 \cdot \epsilon_{N+1} \\
\hat{z}_f^e &= B\left(f^{\text{min-range}}\left(1/(\hat{y}_f^r)^2\right)\right) \cdot \hat{y}_f^e + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z \\
&= B\left(f^{\text{min-range}}\left(1/(\hat{y}_f^r)^2\right)\right) \cdot B(\hat{y}_f^r) \cdot u \cdot \epsilon_y + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z,
\end{aligned}$$

B AA-Based Error Model for Floating-Point Square Root

Given the AA expression for a floating-point variable x_f as

$$\begin{aligned}\hat{x}_f &= \hat{x}_f^r + \hat{x}_f^e \\ &= (x_0 + \sum_{i=1}^{N} x_i \epsilon_i) + B(\hat{x}_f^r) \cdot u \cdot \epsilon_x,\end{aligned}$$

we would like to evaluate the square root of this variable: $\hat{z}_f = \sqrt{\hat{x}_f}$ such that the range component and rounding error component of $\hat{z}_f$ can be represented via the range component and rounding error component of $\hat{x}_f$ as

$$\hat{z}_f = \sqrt{\hat{x}_f^r + \hat{x}_f^e} = \hat{z}_f^r + \hat{z}_f^e.$$

We rewrite the above expression as follows:

$$\begin{aligned}\hat{z}_f &= \sqrt{\hat{x}_f^r + \hat{x}_f^e} \\ &= \sqrt{\hat{x}_f^r \left(1 + \frac{\hat{x}_f^e}{\hat{x}_f^r}\right)} \\ &= \sqrt{\hat{x}_f^r} \cdot \sqrt{1 + \frac{\hat{x}_f^e}{\hat{x}_f^r}}.\end{aligned}$$

Using the Taylor series for approximating $\sqrt{1+x}$, where $x < 1$, and taking only the first order terms of the Taylor series give us

$$\sqrt{1+x} \approx 1 + \frac{1}{2}x, \quad x < 1.$$

Because of $\frac{\hat{x}_f^e}{\hat{x}_f^r} < 1$, the Taylor approximation above can be applied to the approximation of $\sqrt{1 + \frac{\hat{x}_f^e}{\hat{x}_f^r}}$ as follows

$$\sqrt{1 + \frac{\hat{x}_f^e}{\hat{x}_f^r}} \approx 1 + \frac{1}{2} \cdot \frac{\hat{x}_f^e}{\hat{x}_f^r}.$$

We are now ready to evaluate the range component and rounding error component of $\hat{z}_f$.

$$\hat{z}_f \quad = \quad \sqrt{\hat{x}_f^r} \cdot \sqrt{1 + \frac{\hat{x}_f^e}{\hat{x}_f^r}} \approx \sqrt{\hat{x}_f^r} \cdot \left(1 + \frac{1}{2} \cdot \frac{\hat{x}_f^e}{\hat{x}_f^r}\right) = \sqrt{\hat{x}_f^r} + \frac{1}{2\sqrt{\hat{x}_f^r}} \cdot \hat{x}_f^e$$

From this, the range component and the rounding error component of $\hat{z}_f = \sqrt{\hat{x}_f}$ can be described as follows:

$$\begin{aligned} \hat{z}_f^r &= \sqrt{\hat{x}_f^r} \\ \hat{z}_f^e &= \frac{1}{2\sqrt{\hat{x}_f^r}} \cdot \hat{x}_f^e + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z, \end{aligned}$$

where a new error term $B(\hat{z}_f^r) \cdot u \cdot \epsilon_z$ associated with a new noise term ϵ_z is introduced in the rounding error component to represent the rounding error of the square root itself. The remaining error term in the rounding error component $\hat{z}_f^e$ comes from the input operand x, i.e., $\hat{x}_f^e$.

The expressions for the range and error components of $\hat{z}_f$ shown above are not affine forms. In order to have the affine representations for the range and error components of $\hat{z}_f$, we use the *Chebyshev approximation* and the *min-range approximation* [53], as well as apply the bounding operator $B(\cdot)$. Note that we replace $\sqrt{\hat{x}_f^r}$ in the error component by $\hat{z}_f^r$.

$$\begin{aligned} \hat{z}_f^r &= f^{\text{Chebyshev}}\left(\sqrt{\hat{x}_f^r}\right) \\ &= \alpha \cdot \hat{x}_f^r + C_0 + C_1 \cdot \epsilon_{N+1} \\ &= (\alpha \cdot x_0 + C_0) + \sum_{i=1}^{N} x_i \epsilon_i + C_1 \cdot \epsilon_{N+1} \\ \hat{z}_f^e &= \frac{1}{2} B\left(f^{\text{min-range}}\left(1/\hat{z}_f^r\right)\right) \cdot \hat{x}_f^e + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z. \\ &= \frac{1}{2} B\left(f^{\text{min-range}}\left(1/\hat{z}_f^r\right)\right) \cdot B(\hat{x}_f^r) \cdot u \cdot \epsilon_x + B(\hat{z}_f^r) \cdot u \cdot \epsilon_z. \end{aligned}$$

The evaluation of all the coefficients of the Chebyshev and min-range approximations are presented in Section 3.3.2. Extensive discussions on the Chebyshev and min-range approximations are given in [53].

C Norms and Condition Number

This appendix is based on Chapter 6 in [5] and Chapter 2 in [64].

Norms are an indispensable tool in numerical linear algebra. Their purpose is to compress the mn numbers of an $m \times n$ matrix into a single scalar measure of size allowing for a concise and easily interpretable form for perturbation and rounding error analysis.

C.1 Vector Norms

Given real column vectors $\mathbf{x}$ and $\mathbf{y}$ of length n: $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, and a real scalar $\alpha \in \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers and $\mathbb{R}^n$ is the n-dimensional space. A vector norm is a function $||\cdot|| : \mathbb{R}^n \to \mathbb{R}$ satisfying the following conditions:

1. $||\mathbf{x}|| \geq 0$ with equality iff $\mathbf{x} = 0$.
2. $||\alpha\mathbf{x}|| = |\alpha|\ ||\mathbf{x}||$ for all $\alpha \in \mathbb{R},\ \mathbf{x} \in \mathbb{R}^n$.
3. $||\mathbf{x}+\mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}||$ for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$.

A useful class of vector norms are the p-norms defined by

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{\frac{1}{p}} = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}} \qquad p \geq 1, \tag{C.1}$$

of these the three most important vector norms in error analysis and numerical computations are the 1-norm, 2-norm and ∞-norm defined as

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i|, \tag{C.2}$$

$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2} = \sqrt{\mathbf{x}^T\mathbf{x}}, \tag{C.3}$$

$$||\mathbf{x}||_\infty = \max_{1\leq i\leq n} |x_i|. \tag{C.4}$$

C.2 Matrix Norms

A matrix norm is a function $||\cdot|| : \mathbb{R}^{m\times n} \to \mathbb{R}$ satisfying properties analoguos to the three vector norm properties. The simplest example is the Frobenius norm defined by

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}. \tag{C.5}$$

The matrix p-norms are defined in terms of the vector p-norms. The matrix 1-norm and ∞-norm are defined as

$$||\mathbf{A}||_1 = \max_{1\le j\le n}\sum_{i=1}^{n}|a_{ij}|, \tag{C.6}$$

$$||\mathbf{A}||_\infty = \max_{1\le i\le n}\sum_{j=1}^{n}|a_{ij}|. \tag{C.7}$$

Therefore, the matrix 1-norm is called *"max column sum"* and the matrix ∞-norm is called *"max row sum"*. To remember the formulae for the 1-norm and ∞-norm, note that 1 is a vertical symbol (for columns) and ∞ is a horizontal symbol (for rows).

The p-norms have the important property that for every matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$ we have [64]

$$||\mathbf{Ax}||_p \le ||\mathbf{A}||_p||\mathbf{x}||_p. \tag{C.8}$$

C.3 Condition Number

We analyse the linear system

$$\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$ is an $n \times n$ square matrix and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$ are column vectors of length n. Assuming that matrix $\mathbf{A}$ is invertible (or non-singular) means its inverse matrix $\mathbf{A^{-1}}$ exists. The condition number $\kappa(\mathbf{A})$ of matrix $\mathbf{A}$ is defined to be [64]

$$\kappa(\mathbf{A}) = ||\mathbf{A}||\cdot||\mathbf{A}^{-1}||, \tag{C.9}$$

where $||\cdot||$ is some matrix norm. In this thesis the 1-norm is used to compute the condition number. The condition number satisfies $\kappa(\mathbf{A}) \ge 1$ and can be arbitrarily large. By convention, the condition number $\kappa(\mathbf{A}) = \infty$ if $\mathbf{A}$ is not invertible.

Assuming matrix $\mathbf{A}$ is perturbed by $\mathbf{\Delta A}$ and vector $\mathbf{b}$ is perturbed by $\mathbf{\Delta b}$.The perturbed linear system of equations will be

$$(\mathbf{A}+\mathbf{\Delta A})\cdot\hat{\mathbf{x}} = (\mathbf{b}+\mathbf{\Delta b})$$

where

$$\hat{\mathbf{x}} = \mathbf{x}+\mathbf{\Delta x}$$

is the perturbed solution. The condition number $\kappa(\mathbf{A})$ quantifies the sensitivity of the solution of linear system to the perturbation in $\mathbf{A}$ and $\mathbf{b}$ by the following relation [64]

$$\frac{||\mathbf{\Delta x}||}{||\mathbf{x}||} \le \kappa(\mathbf{A})\left(\frac{||\mathbf{\Delta A}||}{||\mathbf{A}||}+\frac{||\mathbf{\Delta b}||}{||\mathbf{b}||}\right), \tag{C.10}$$

such that the relative error in $\mathbf{x}$ can be $\kappa(\mathbf{A})$ times the relative error in $\mathbf{A}$ and $\mathbf{b}$. The condition number gives a bound on how inaccurate the solution $\mathbf{x}$ will be after approximating the solution. Note that this is before the effects of rounding errors are taken into account. Therefore, conditioning is a property of the matrix $\mathbf{A}$, neither of the algorithm nor of the floating point number format used to solve the corresponding linear system. If the condition number is "small", the linear system is referred to as well-conditioned, otherwise it is called an ill-conditioned system. "Small" condition number basically means $\kappa(\mathbf{A})$ is close to 1.

D
Matlab Code

D.1 Code for custom-precision floating-point Levinson-Durbin algorithm

We implemented in Matlab the custom-precision implementation version of Levinson-Durbin algorithm, namely `levcus`, by incorporating MPFR functions as .MEX files.

```
function [a,E,k, findex] = levcus(r, n, prec)
% LEVCUS Levinson-Durbin iteration in Custom-Precision Floating-Point Format
%   [a,E,k, findex] = levcus(r, n, prec)
%   This function implement the Levinson-Durbin alogorithm to find the
%   optimal coefficients A of one wide-sense stationary process by using
%   auto-correlation method
%       [  R(1)    R(2)   ...   R(N)   ] [   A(2)   ]  = [  -R(2)   ]
%       [  R(2)    R(1)   ... R(N-1)   ] [   A(3)   ]  = [  -R(3)   ]
%       [    .        .            .   ] [    .     ]  = [     .    ]
%       [ R(N-1) R(N-2)   ...   R(2)   ] [   A(N)   ]  = [  -R(N)   ]
%       [  R(N)  R(N-1)   ...   R(1)   ] [ A(N+1)   ]  = [  -R(N+1)]
%
%   r    : INPUT column vector: Auto-correlation input vector
%   n    : INPUT scalar value:  Order of predictor with default value n= length(r)-1
%   prec: INPUT scalar value:  working precision (5 <= prec <= 53)
%
%   a:  OUTPUT column vector: Optimal output coefficients
%   k:  OUTPUT column vector: Reflection coefficients
%   E:  OUTPUT column vector: contains the energy of prediction error at each
    iteration
%   findex: OUTPUT scalar value: indicates the failing iteration at which |k| > 1!
%           when |k|>1 the Levinson-Durbin iteration stops and findex is reported:
%       - findex = 0          if Levinson-Durbin was processed successfully,
%       - findex = iteration at that the Levinson-Durbin was failed (i.e., |k| > 1).
%
%   Reference:  Peter Vary & Rainer Martin, 2006
%               "Digital Speech Transmission" page 182-184
%   Update: Oct 05th, 2011
%   By Thang Viet Huynh, SPSC, TU Graz <thang.huynhviet@tugraz.at>

%% Check number of input arguments
switch nargin
```

```
    case 1
        n=length(r)-1;    % order of the predictor
        prec = 53;        %default is double-precision
    case 2
        prec = 53;        %default is double-precision
    case 3
        % do nothing!
    otherwise
        error('The correct call to Levinson-Durbin algorithm is as [a,E,k,findex] =
            levcus(r,n,prec).');
end


%% Pre-allocate variables
a = zeros(n+1,1);
a(1)=1;

k = zeros(n+1,1);
k(1)=1;

E = zeros(n+1,1);
E(1)=r(1);

findex = 0;


% Declare a copy of vector a
% to store the previous value of a
% af is also a flipped copy of a
af = a;


%% ---------------------
% Recursive-Loop
%-----------------------
% This custom-precision floating-point computation is based on the MPFR
% Library 3.0, with the following functions performing custom-precision
% floating-point operations in Matlab (via .MEX files):
%
% - z = add_mpfr (x, prec, y, prec) --> z = x + y
%
% - z = mul_mpfr (x, prec, y, prec) --> z = x * y
%
% - z = div_mpfr (x, prec, y, prec) --> z = x / y
%
% - z = fma_mpfr (x, prec, y, prec, w, prec) --> z = x * y + w


for p = 1:n

    % Step 1
    q = 0;
    for i=1:p
        %q = q + a(i)*r(p-i+2); % original expression in SP or DP
        q = add_mpfr (q, prec, mul_mpfr(a(i),prec,r(p-i+2),prec), prec);
    end

    % Step 2
    %k(p+1) = -q/E(p); % original expression in SP or DP
    k(p+1) = -div_mpfr(q, prec, E(p), prec);

    % check if |k| <= 1
    if (abs(k(p+1))>1)
        findex = p;
        break;  % return
    end

    % Step 3  % update a
    for i=1:p+1
        af(i)=a(p-i+2); %update af vector
    end

    for i=2:(p+1)
        %a(i) = a(i) + k(p+1)*af(i); % original expression in SP or DP
        a(i) = add_mpfr(a(i), prec, mul_mpfr(k(p+1),prec,af(i),prec), prec);
    end

    % Step 4
    %E(p+1)=E(p)*(1-(k(p+1)*k(p+1))); % original expression in SP or DP
    one_minus_k2 = add_mpfr(1, prec, - mul_mpfr(k(p+1),prec,k(p+1),prec), prec);
    E(p+1) = mul_mpfr(E(p), prec, one_minus_k2, prec);

end
```

D.2 Code for AA-based floating-point error analysis of the Levinson-Durbin algorithm using the `AAFloat` class

```
% AA-based error analysis for the Levinson-Durbin algorithm using the AAFloat class
% update 05.04.2012

clear all;
implementation_name = 'Levinson-Durbin algorithm for special case [0.17, 0.23]';
system_order = 10;
a1 = 0.17;
b1 = 0.23;
p = 24;


% declare some constants
u = 2^(-p);
DEFAULT_PROB   = 0.997300203937;
AA_HARD_BOUND = 1.0;                AAText_HARD = 'Hard';
AA_SOFT_BOUND = DEFAULT_PROB;       AAText_SOFT = 'Soft';

% specify the bounding operator used
% prob = AA_HARD_BOUND; AAText = AAText_HARD;
```

```matlab
prob = AA_SOFT_BOUND;  AAText = AAText_SOFT;

%% declare AA forms for some constants
tic;
AAFloat_INIT;
ONE    = AAFloat(1,1,p);
AA_a0 = ONE;                    % a0 = 1
AA_k0 = ONE;                    % k0 = 1

% declare AA form for autocorrelation coeff. r
AAFloat_INIT;
autocoeff = AAFloat(a1*ones(1,system_order), b1*ones(1,system_order), p);
autocoeff = [ONE, autocoeff];

a = AA_a0;
k = AA_k0;

AAFloat_INIT;
LD_alpha = ONE;        % alpha0 = r0

% Variables for trace of rounding errors
AAFloat_INIT;
A             = AAFloat(zeros(system_order+1,system_order+1), zeros(system_order+1,
    system_order+1), p);
for m=1:system_order+1
    A(m,1) = ONE;
end
LD_BETA       = AAFloat(zeros(system_order+1,1), zeros(system_order+1,1), p);
LD_ALPHA      = AAFloat(zeros(system_order+1,1), zeros(system_order+1,1), p);

%%
AAFloat_INIT;
fprintf('\n*****************************************************************\n');
fprintf('*****************************************************************\n');

AA_bound_a = zeros(system_order, system_order);
AA_bound_k = zeros(system_order, 1);

for m=1 : system_order

    % Execute the Levinson-Durbin algorithm
    fprintf('\n---------- Iteration m = %d\n',m);
    fprintf('%s\n\n', implementation_name);
    fprintf('AA bound = %s, precision = %d, r in [%5.2f,%5.2f]\n', AAText, p, a1, b1);

    % LD_alpha before updated
    temp = ebound(LD_alpha, prob);
    fprintf('error bound of E_{m-1} = %e', temp);
    temp = aa2ia(LD_alpha, prob);
    fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));
    fprintf('\n');

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Begin of LEVINSON-DURBIN Algorithm
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Step 1: compute the numerator beta
    r = autocoeff(2:m+1);
    LD_beta = mmul(r, (fliplr(a))', prob);

    % Step 2: compute reflection coefficient k
    kk = -div(LD_beta, LD_alpha, prob);                % k = -beta/alpha
    k  = [k, kk];                                      % add new reflection coeff.

    % Step 3a: update filter coefficients a
    a = [a, kk];
    a_flip = fliplr(a);
    a_temp = AA_a0;
    for i=2 : ( length(a)-1)
        ka = mul(kk, a_flip(i), prob);                 % k*a(m-i)
        a_update = add(a(i), ka, prob);                % a(i) = a(i) + k*a(m-i)
        a_temp = [a_temp, a_update];
    end
    a_temp   = [a_temp, kk];
    a        = a_temp;

    % Step 3b: Update alpha
    kk2          = sqr(kk, prob);                     % k*k
    kk2_1        = sub(1, kk2, prob);                 % 1 - k*k
    LD_alpha     = mul(LD_alpha, kk2_1, prob);        % E = E(1-k*k)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% End of LEVINSON-DURBIN Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % save all variables at the current iteration
  % save A
  A(m+1,1:m+1) = a;

  % save BETA
  LD_BETA(m+1) = LD_beta;

  % save ALPHA
  LD_ALPHA(m+1) = LD_alpha;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% Display error bound at each iteration

  % beta
  temp = ebound(LD_beta, prob);
  fprintf('error bound of beta  = %e', temp);
  temp = aa2ia(LD_beta, prob);
  fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));
  fprintf('\n');

  % display error bound for reflection coefficient
  for i=1: length(k),
      temp = ebound(k(i), prob);
      AA_bound_k(i) = temp;
      fprintf('i=%2d, error bound of k(%2d) = %e', i, i, temp);
      temp = aa2ia(k(i),prob);
      fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));
  end;
```

```
    fprintf('\n');

    % display error bound for filter coefficient
    for i=1: length(a),
        temp = ebound(a(i), prob);
        AA_bound_a(m, i) = temp;
        fprintf('i=%2d, error bound of a(%2d) = %e', i, i, temp);
        temp = aa2ia(a(i),prob);
        fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));
    end;
    fprintf('\n');

    % display error bound for short-term prediction error
    temp = ebound(LD_alpha, prob);
    fprintf('error bound of E_{m} = %e', temp);
    temp = aa2ia(LD_alpha,prob);
    fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));
    fprintf('\n');

end;
t = toc; t = t/60;

fprintf('\n');
fprintf('Time = %f (minutes)\n',t);

K = k;
clear a k LD_beta LD_alpha AA_a0 AA_k0

datafile = ['LevError_',AAText,'_Bound_SimpleExample_Order',num2str(system_order),'
    _prec',num2str(p)];
save(datafile);

return;
```

D.3 Code for floating-point error analysis of Levinson-Durbin algorithm using AA-based Scaling Operator

```
% AA-based error analysis for the Levinson-Durbin algorithm using the AAFloat class
% Apply the AA-based Scaling Operator
% update 05.04.2012

clear all;

implementation_name = 'Levinson-Durbin algorithm for general range [-1,+1] using range
    -scaling';
system_order = 10;
a1 = -1;
b1 = +1;
p = 24;

delta_E  = 0.0001;

```

```
% declare some constants
u = 2^(-p);
DEFAULT_PROB  = 0.997300203937;
AA_HARD_BOUND = 1.0;              AAText_HARD = 'Hard';
AA_SOFT_BOUND = DEFAULT_PROB;     AAText_SOFT = 'Soft';

%%% specify the bounding operator used
prob = AA_HARD_BOUND; AAText = AAText_HARD;
% prob = AA_SOFT_BOUND; AAText = AAText_SOFT;

%% declare AA forms for some constants
tic;
AAFloat_INIT;
ONE    = AAFloat(1,1,p);
AA_a0 = ONE;                    % a0 = 1
AA_k0 = ONE;                    % k0 = 1

% declare AA form for autocorrelation coeff. r
AAFloat_INIT;
autocoeff = AAFloat(a1*ones(1,system_order), b1*ones(1,system_order), p);
autocoeff = [ONE, autocoeff];

a = AA_a0;
k = AA_k0;

AAFloat_INIT;
LD_alpha = ONE;        % alpha0 = r0

% Variables for trace of rounding errors
AAFloat_INIT;
A            = AAFloat(zeros(system_order+1,system_order+1), zeros(system_order+1,
    system_order+1), p);
for m=1:system_order+1
    A(m,1) = ONE;
end
LD_BETA      = AAFloat(zeros(system_order+1,1), zeros(system_order+1,1), p);
LD_ALPHA     = AAFloat(zeros(system_order+1,1), zeros(system_order+1,1), p);

%%
AAFloat_INIT;
fprintf('\n**********************************************************************\n');
fprintf('**********************************************************************\n');

AA_bound_a = zeros(system_order, system_order);
AA_bound_k = zeros(system_order, 1);

for m=1 : system_order

    % Execute the Levinson-Durbin algorithm
    fprintf('\n---------- Iteration m = %d\n',m);
    fprintf('%s\n\n', implementation_name);
    fprintf('AA bound = %s, precision = %d, r in [%5.2f,%5.2f]\n', AAText, p, a1, b1);
    fprintf('delta    = %f\n', delta);

    % LD_alpha before updated
    temp = ebound(LD_alpha, prob);
    fprintf('error bound of E_{m-1} = %e', temp);
    temp = aa2ia(LD_alpha, prob);
    fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Begin of LEVINSON-DURBIN Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Step 1: compute the numerator beta
    r = autocoeff(2:m+1);
    LD_beta = mmul(r, (fliplr(a))', prob);

    % 1a: convert LD_beta into the range [0, 2^{m-1}]
    LD_beta = AASO(LD_beta, [-1, 1]);

    % display beta
    temp = ebound(LD_beta, prob);
    fprintf('error bound of beta    = %e', temp);
    temp = aa2ia(LD_beta, prob);
    fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));
    fprintf('\n');


    % Step 2: compute reflection coefficient k
    kk = -div(LD_beta, LD_alpha, prob);              % beta/alpha
    kk = AASO(kk, [-1, 1]);
    k  = [k, kk];                                    % add new reflection coeff.


    % Step 3a: update filter coefficients a
    a = [a, kk];
    a_flip = fliplr(a);
    a_temp = AA_a0;
    for i=2 : ( length(a)-1)
        ka = mul(kk, a_flip(i), prob);  % k*a(m-i)
        a_update = add(a(i), ka, prob);   % a(i) = a(i) + k*a(m-i)
        a_temp = [a_temp, a_update];
    end
    a_temp  = [a_temp, kk];
    a       = a_temp;

    % Step 3b: Update alpha
    kk2         = sqr(kk, prob);                    % k*k
    kk2_1       = sub(1, kk2, prob);                % 1 - k*k
    LD_alpha    = mul(LD_alpha, kk2_1, prob);       % E = E(1-k*k)
    % convert LD_alpha into the range [\delta, 2^{m-1}]
    LD_alpha = AASO (LD_alpha, [delta, 1]);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% End of LEVINSON-DURBIN Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % save all variables at the current iteration
    % save A
    A(m+1,1:m+1) = a;

    % save BETA
    LD_BETA(m+1) = LD_beta;

    % save ALPHA
    LD_ALPHA(m+1) = LD_alpha;

    %% Display error bound at each iteration
    fprintf('\n');

    % display error bound for reflection coefficient
    for i=2: length(k),
        temp = ebound(k(i), prob);
        AA_bound_k(i) = temp;
        fprintf('i=%2d, error bound of k(%2d) = %e', i-1, i-1, temp);
        temp = aa2ia(k(i),prob);
        fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));
    end;
    fprintf('\n');

    % display error bound for filter coefficient
    for i=1: length(a),
        temp = ebound(a(i), prob);
        AA_bound_a(m, i) = temp;
        fprintf('i=%2d, error bound of a(%2d) = %e', i-1, i-1, temp);
        temp = aa2ia(a(i), prob);
        fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));
    end;
    fprintf('\n');

    % display error bound for short-term prediction error
    temp = ebound(LD_alpha, prob);
    fprintf('error bound of E_{m} = %e', temp);
    temp = aa2ia(LD_alpha, prob);
    fprintf(', range = [%f, %f]\n', temp.r(1), temp.r(2));
    fprintf('\n');

end;
t = toc; t = t/60;

fprintf('\n');
fprintf('Time = %f (minutes)\n',t);

K = k;
clear a k LD_beta LD_alpha AA_a0 AA_k0

datafile = ['LevError_',AAText,'_Bound_GeneralRangeCorrection_Order',num2str(
    system_order),'_prec',num2str(p)];
save(datafile);

return;
```

E

Reports for Error Analysis of Levinson-Durbin Algorithm using AAFloat Tool

E.1 Report for scenarios 1 and 2

The running report for the scenario 1 corresponds to iterations 1 to 10. The running report for the scenario 2 includes all iterations.

```
*****************************************************************
*****************************************************************

————————— Iteration m = 1
Levinson−Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m−1} = 0.000000e+00, range = [1.000000, 1.000000]
error bound of beta    = 2.741814e−08, range = [0.170000, 0.230000]


i= 1, error bound of k( 1) = 5.483627e−08, range = [−0.230000, −0.170000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 5.483627e−08, range = [−0.230000, −0.170000]

error bound of E_{m} = 1.302063e−07, range = [0.947100, 0.972900]


————————— Iteration m = 2
Levinson−Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m−1} = 1.302063e−07, range = [0.947100, 0.972900]
error bound of beta    = 4.581809e−08, range = [0.117100, 0.202900]


i= 1, error bound of k( 1) = 5.483627e−08, range = [−0.230000, −0.170000]
i= 2, error bound of k( 2) = 1.017001e−07, range = [−0.210176, −0.123218]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
```

```
i= 1, error bound of a( 1) = 1.006850e−07, range = [−0.201660, −0.131661]
i= 2, error bound of a( 2) = 1.017001e−07, range = [−0.210176, −0.123218]

error bound of E_{m} = 2.177255e−07, range = [0.912108, 0.954549]


−−−−−−−−−−− Iteration m = 3
Levinson−Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m−1} = 2.177255e−07, range = [0.912108, 0.954549]
error bound of beta    = 8.074333e−08, range = [0.082108, 0.184549]


i= 1, error bound of k( 1) = 5.483627e−08, range = [−0.230000, −0.170000]
i= 2, error bound of k( 2) = 1.017001e−07, range = [−0.210176, −0.123218]
i= 3, error bound of k( 3) = 1.594273e−07, range = [−0.196811, −0.089042]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.456357e−07, range = [−0.192202, −0.093469]
i= 2, error bound of a( 2) = 1.365928e−07, range = [−0.199575, −0.086178]
i= 3, error bound of a( 3) = 1.594273e−07, range = [−0.196811, −0.089042]

error bound of E_{m} = 3.121425e−07, range = [0.885643, 0.942902]


−−−−−−−−−−− Iteration m = 4
Levinson−Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m−1} = 3.121425e−07, range = [0.885643, 0.942902]
error bound of beta    = 1.245731e−07, range = [0.055643, 0.172902]


i= 1, error bound of k( 1) = 5.483627e−08, range = [−0.230000, −0.170000]
i= 2, error bound of k( 2) = 1.017001e−07, range = [−0.210176, −0.123218]
i= 3, error bound of k( 3) = 1.594273e−07, range = [−0.196811, −0.089042]
i= 4, error bound of k( 4) = 2.314679e−07, range = [−0.189101, −0.061118]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.819867e−07, range = [−0.186580, −0.063328]
i= 2, error bound of a( 2) = 1.817985e−07, range = [−0.183114, −0.066889]
i= 3, error bound of a( 3) = 1.897625e−07, range = [−0.191003, −0.059110]
i= 4, error bound of a( 4) = 2.314679e−07, range = [−0.189101, −0.061118]

error bound of E_{m} = 4.099125e−07, range = [0.863813, 0.936138]


−−−−−−−−−−− Iteration m = 5
Levinson−Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m−1} = 4.099125e−07, range = [0.863813, 0.936138]
error bound of beta    = 1.711859e−07, range = [0.033982, 0.165970]


```

```
i= 1, error bound of k( 1) = 5.483627e−08, range = [−0.230000, −0.170000]
i= 2, error bound of k( 2) = 1.017001e−07, range = [−0.210176, −0.123218]
i= 3, error bound of k( 3) = 1.594273e−07, range = [−0.196811, −0.089042]
i= 4, error bound of k( 4) = 2.314679e−07, range = [−0.189101, −0.061118]
i= 5, error bound of k( 5) = 3.109584e−07, range = [−0.185614, −0.036920]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.116300e−07, range = [−0.184987, −0.037079]
i= 2, error bound of a( 2) = 2.218520e−07, range = [−0.179995, −0.042179]
i= 3, error bound of a( 3) = 2.024525e−07, range = [−0.188294, −0.034002]
i= 4, error bound of a( 4) = 2.579072e−07, range = [−0.187487, −0.034927]
i= 5, error bound of a( 5) = 3.109584e−07, range = [−0.185614, −0.036920]

error bound of E_{m} = 5.085796e−07, range = [0.843477, 0.934226]


———————— Iteration m = 6
Levinson−Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m−1} = 5.085796e−07, range = [0.843477, 0.934226]
error bound of beta    = 2.237670e−07, range = [0.013486, 0.164218]


i= 1, error bound of k( 1) = 5.483627e−08, range = [−0.230000, −0.170000]
i= 2, error bound of k( 2) = 1.017001e−07, range = [−0.210176, −0.123218]
i= 3, error bound of k( 3) = 1.594273e−07, range = [−0.196811, −0.089042]
i= 4, error bound of k( 4) = 2.314679e−07, range = [−0.189101, −0.061118]
i= 5, error bound of k( 5) = 3.109584e−07, range = [−0.185614, −0.036920]
i= 6, error bound of k( 6) = 4.054436e−07, range = [−0.187718, −0.012729]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.371997e−07, range = [−0.187688, −0.012076]
i= 2, error bound of a( 2) = 2.545590e−07, range = [−0.182601, −0.017281]
i= 3, error bound of a( 3) = 2.647884e−07, range = [−0.180427, −0.019590]
i= 4, error bound of a( 4) = 2.776551e−07, range = [−0.189028, −0.011118]
i= 5, error bound of a( 5) = 3.423762e−07, range = [−0.188213, −0.012064]
i= 6, error bound of a( 6) = 4.054436e−07, range = [−0.187718, −0.012729]

error bound of E_{m} = 6.092186e−07, range = [0.822642, 0.937252]


———————— Iteration m = 7
Levinson−Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m−1} = 6.092186e−07, range = [0.822642, 0.937252]
error bound of beta    = 2.731665e−07, range = [−0.007290, 0.167183]


i= 1, error bound of k( 1) = 5.483627e−08, range = [−0.230000, −0.170000]
i= 2, error bound of k( 2) = 1.017001e−07, range = [−0.210176, −0.123218]
i= 3, error bound of k( 3) = 1.594273e−07, range = [−0.196811, −0.089042]
i= 4, error bound of k( 4) = 2.314679e−07, range = [−0.189101, −0.061118]
i= 5, error bound of k( 5) = 3.109584e−07, range = [−0.185614, −0.036920]
i= 6, error bound of k( 6) = 4.054436e−07, range = [−0.187718, −0.012729]
```

```
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.616557e-07, range = [-0.195717, 0.014243]
i= 2, error bound of a( 2) = 2.848413e-07, range = [-0.188962, 0.007353]
i= 3, error bound of a( 3) = 3.080051e-07, range = [-0.187698, 0.005942]
i= 4, error bound of a( 4) = 2.935171e-07, range = [-0.194346, 0.012449]
i= 5, error bound of a( 5) = 3.650987e-07, range = [-0.194408, 0.012368]
i= 6, error bound of a( 6) = 4.403941e-07, range = [-0.194990, 0.012770]
i= 7, error bound of a( 7) = 5.059894e-07, range = [-0.195617, 0.013135]

error bound of E_{m} = 7.149309e-07, range = [0.798357, 0.946947]


----------- Iteration m = 8
Levinson-Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 7.149309e-07, range = [0.798357, 0.946947]
error bound of beta    = 3.307788e-07, range = [-0.031643, 0.176947]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 3.012806e-07, range = [-0.211312, 0.045142]
i= 2, error bound of a( 2) = 3.195183e-07, range = [-0.203414, 0.037086]
i= 3, error bound of a( 3) = 3.483065e-07, range = [-0.201084, 0.034596]
i= 4, error bound of a( 4) = 3.810710e-07, range = [-0.196403, 0.029761]
i= 5, error bound of a( 5) = 3.803162e-07, range = [-0.205995, 0.039198]
i= 6, error bound of a( 6) = 4.718850e-07, range = [-0.207282, 0.040292]
i= 7, error bound of a( 7) = 5.589269e-07, range = [-0.208989, 0.041726]
i= 8, error bound of a( 8) = 6.386663e-07, range = [-0.213076, 0.045351]

error bound of E_{m} = 8.339088e-07, range = [0.766725, 0.966394]


----------- Iteration m = 9
Levinson-Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 8.339088e-07, range = [0.766725, 0.966394]
error bound of beta    = 4.001061e-07, range = [-0.063275, 0.196394]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
```

```
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e-07, range = [-0.246109, 0.090425]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 3.841044e-07, range = [-0.239477, 0.086362]
i= 2, error bound of a( 2) = 3.764219e-07, range = [-0.229079, 0.075772]
i= 3, error bound of a( 3) = 3.912121e-07, range = [-0.225934, 0.072444]
i= 4, error bound of a( 4) = 4.489043e-07, range = [-0.219977, 0.066318]
i= 5, error bound of a( 5) = 4.106854e-07, range = [-0.226365, 0.072539]
i= 6, error bound of a( 6) = 4.862096e-07, range = [-0.229658, 0.075628]
i= 7, error bound of a( 7) = 6.073359e-07, range = [-0.232115, 0.077800]
i= 8, error bound of a( 8) = 7.188294e-07, range = [-0.237448, 0.082658]
i= 9, error bound of a( 9) = 8.302704e-07, range = [-0.246109, 0.090425]

error bound of E_{m} = 9.818680e-07, range = [0.718425, 1.004332]


---------- Iteration m = 10
Levinson-Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 9.818680e-07, range = [0.718425, 1.004332]
error bound of beta    = 4.990450e-07, range = [-0.111575, 0.234332]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e-07, range = [-0.246109, 0.090425]
i=10, error bound of k(10) = 1.178539e-06, range = [-0.313863, 0.167315]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 5.677371e-07, range = [-0.293483, 0.151776]
i= 2, error bound of a( 2) = 5.017623e-07, range = [-0.278191, 0.136226]
i= 3, error bound of a( 3) = 4.695504e-07, range = [-0.272332, 0.130149]
i= 4, error bound of a( 4) = 5.318313e-07, range = [-0.264763, 0.122391]
i= 5, error bound of a( 5) = 5.752593e-07, range = [-0.256556, 0.114001]
i= 6, error bound of a( 6) = 5.191886e-07, range = [-0.270415, 0.127644]
i= 7, error bound of a( 7) = 6.537136e-07, range = [-0.275206, 0.132137]
i= 8, error bound of a( 8) = 7.919249e-07, range = [-0.281671, 0.138114]
i= 9, error bound of a( 9) = 9.624451e-07, range = [-0.291951, 0.147487]
i=10, error bound of a(10) = 1.178539e-06, range = [-0.313863, 0.167315]

error bound of E_{m} = 1.200439e-06, range = [0.630733, 1.083029]


---------- Iteration m = 11
Levinson-Durbin algorithm for special case [0.17, 0.23]
```

```

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m−1} = 1.200439e−06, range = [0.630733, 1.083029]
error bound of beta    = 6.677230e−07, range = [−0.199176, 0.312938]


i= 1, error bound of k( 1) = 5.483627e−08, range = [−0.230000, −0.170000]
i= 2, error bound of k( 2) = 1.017001e−07, range = [−0.210176, −0.123218]
i= 3, error bound of k( 3) = 1.594273e−07, range = [−0.196811, −0.089042]
i= 4, error bound of k( 4) = 2.314679e−07, range = [−0.189101, −0.061118]
i= 5, error bound of k( 5) = 3.109584e−07, range = [−0.185614, −0.036920]
i= 6, error bound of k( 6) = 4.054436e−07, range = [−0.187718, −0.012729]
i= 7, error bound of k( 7) = 5.059894e−07, range = [−0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e−07, range = [−0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e−07, range = [−0.246109, 0.090425]
i=10, error bound of k(10) = 1.178539e−06, range = [−0.313863, 0.167315]
i=11, error bound of k(11) = 2.061141e−06, range = [−0.480281, 0.337578]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.003311e−06, range = [−0.420612, 0.289361]
i= 2, error bound of a( 2) = 9.741858e−07, range = [−0.391389, 0.259732]
i= 3, error bound of a( 3) = 8.470351e−07, range = [−0.375696, 0.243756]
i= 4, error bound of a( 4) = 8.254623e−07, range = [−0.367075, 0.234911]
i= 5, error bound of a( 5) = 9.307695e−07, range = [−0.355374, 0.223007]
i= 6, error bound of a( 6) = 8.075530e−07, range = [−0.360570, 0.227970]
i= 7, error bound of a( 7) = 8.740120e−07, range = [−0.370919, 0.238009]
i= 8, error bound of a( 8) = 1.048553e−06, range = [−0.381153, 0.247742]
i= 9, error bound of a( 9) = 1.245666e−06, range = [−0.394048, 0.259712]
i=10, error bound of a(10) = 1.405748e−06, range = [−0.420537, 0.284100]
i=11, error bound of a(11) = 2.061141e−06, range = [−0.480281, 0.337578]

error bound of E_{m} = 2.151582e−06, range = [0.420082, 1.285563]


———————— Iteration m = 12
Levinson−Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m−1} = 2.151582e−06, range = [0.420082, 1.285563]
error bound of beta    = 1.568348e−06, range = [−0.409854, 0.515499]


i= 1, error bound of k( 1) = 5.483627e−08, range = [−0.230000, −0.170000]
i= 2, error bound of k( 2) = 1.017001e−07, range = [−0.210176, −0.123218]
i= 3, error bound of k( 3) = 1.594273e−07, range = [−0.196811, −0.089042]
i= 4, error bound of k( 4) = 2.314679e−07, range = [−0.189101, −0.061118]
i= 5, error bound of k( 5) = 3.109584e−07, range = [−0.185614, −0.036920]
i= 6, error bound of k( 6) = 4.054436e−07, range = [−0.187718, −0.012729]
i= 7, error bound of k( 7) = 5.059894e−07, range = [−0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e−07, range = [−0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e−07, range = [−0.246109, 0.090425]
i=10, error bound of k(10) = 1.178539e−06, range = [−0.313863, 0.167315]
i=11, error bound of k(11) = 2.061141e−06, range = [−0.480281, 0.337578]
i=12, error bound of k(12) = 1.016107e−05, range = [−1.206347, 1.039515]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
```

```
i= 1, error bound of a( 1) = 3.774775e-06, range = [-0.968911, 0.849564]
i= 2, error bound of a( 2) = 4.973436e-06, range = [-0.860309, 0.740033]
i= 3, error bound of a( 3) = 4.347825e-06, range = [-0.806564, 0.685830]
i= 4, error bound of a( 4) = 3.956194e-06, range = [-0.783025, 0.661990]
i= 5, error bound of a( 5) = 3.808731e-06, range = [-0.757836, 0.636556]
i= 6, error bound of a( 6) = 4.006316e-06, range = [-0.720752, 0.599213]
i= 7, error bound of a( 7) = 3.711993e-06, range = [-0.751849, 0.629980]
i= 8, error bound of a( 8) = 3.734243e-06, range = [-0.777295, 0.654909]
i= 9, error bound of a( 9) = 4.044038e-06, range = [-0.800614, 0.677284]
i=10, error bound of a(10) = 4.512724e-06, range = [-0.844361, 0.718906]
i=11, error bound of a(11) = 3.358012e-06, range = [-0.932835, 0.801081]
i=12, error bound of a(12) = 1.016107e-05, range = [-1.206347, 1.039515]

error bound of E_{m} = 1.149381e-05, range = [-0.546389, 2.243222]


---------- Iteration m = 13
Levinson-Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 1.149381e-05, range = [-0.546389, 2.243222]
error bound of beta    = 1.086233e-05, range = [-1.376382, 1.473215]

Warning: The input range [-0.546389, 2.243222] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1072
  In AA_bound_levinson_wAAFloat_simple_example_increase_order at 88
The user does not specify any range for the input interval of the inverse operation.
AAFloat is using the default range [1.192093e-07, 2.243222e+00] to compute the inverse
    .

i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e-07, range = [-0.246109, 0.090425]
i=10, error bound of k(10) = 1.178539e-06, range = [-0.313863, 0.167315]
i=11, error bound of k(11) = 2.061141e-06, range = [-0.480281, 0.337578]
i=12, error bound of k(12) = 1.016107e-05, range = [-1.206347, 1.039515]
i=13, error bound of k(13) = 1.191544e+09, range = [-12358221.218054, 11952076.988843]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.437416e+09, range = [-14850247.879990, 14884126.670198]
i= 2, error bound of a( 2) = 1.111514e+09, range = [-11444673.384395, 11471428.849417]
i= 3, error bound of a( 3) = 1.006093e+09, range = [-10350033.222404, 10375509.506310]
i= 4, error bound of a( 4) = 9.539665e+08, range = [-9809606.568093, 9834651.251699]
i= 5, error bound of a( 5) = 9.261817e+08, range = [-9521868.928906, 9546722.165494]
i= 6, error bound of a( 6) = 8.958615e+08, range = [-9209071.109641, 9233819.194089]
i= 7, error bound of a( 7) = 8.588075e+08, range = [-8823044.851202, 8847725.912109]
i= 8, error bound of a( 8) = 9.029951e+08, range = [-9279778.435664, 9304406.981187]
i= 9, error bound of a( 9) = 9.330091e+08, range = [-9589077.095652, 9613655.846475]
i=10, error bound of a(10) = 9.610570e+08, range = [-9877425.943302, 9901943.523795]
```

```
i=11, error bound of a(11) = 1.025096e+09, range = [-10535360.982528, 10559785.600643]
i=12, error bound of a(12) = 1.154501e+09, range = [-11863667.952314, 11887903.841340]
i=13, error bound of a(13) = 1.191544e+09, range = [-12358221.218054, 11952076.988843]

error bound of E_{m} = 3.072350e+09, range = [-27933964.201552, 27914301.953442]


---------- Iteration m = 14
Levinson-Durbin algorithm for special case [0.17, 0.23]

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 3.072350e+09, range = [-27933964.201552, 27914301.953442]
error bound of beta    = 3.072350e+09, range = [-27932550.575950, 27912886.727841]

Warning: The input range [-27933964.201552, 27914301.953442] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1072
  In AA_bound_levinson_wAAFloat_simple_example_increase_order at 88
The user does not specify any range for the input interval of the inverse operation.
AAFloat is using the default range [-2.793396e+07, -1.192093e-07] to compute the
    inverse.
Warning: Possibility of huge rounding error!
> In AAFloat.AAFloat>AAFloat.inv at 1016
  In AAFloat.AAFloat>AAFloat.div at 1072
  In AA_bound_levinson_wAAFloat_simple_example_increase_order at 88
In inverse operation, error component estimation: range of inverse contains ZERO
    [0.000000e+00, 7.803064e+14].
AAFloat is using the range [1.192093e-07, 7.803064e+14] instead. However, huge
    rounding error may result!
range = [0.000000e+00, 7.803064e+14]
??? Error using ==> AAFloat.AAFloat>aa_minrange_inv at 1871
Error: in aa_minrange_inv approximation: LOWER BOUND of input affine interval equals
    ZERO!

Error in ==> AAFloat.AAFloat>AAFloat.inv at 1031
            xsquareinv          = aa_minrange_inv (xsquare_range);

Error in ==> AAFloat.AAFloat>AAFloat.div at 1072
            yy = inv(y, prob);

Error in ==> AA_bound_levinson_wAAFloat_simple_example_increase_order at 88
    kk = -div(LD_beta, LD_alpha, prob);              % beta/alpha
```

E.2 Report for scenario 3

```
**********************************************************************
**********************************************************************

---------- Iteration m = 1
Levinson-Durbin algorithm for case [-1, 1]

```

```
AA bound = Hard, precision = 24, r in [−1.00, 1.00]
error bound of E_{m−1} = 0.000000e+00, range = [1.000000, 1.000000]
error bound of beta    = 1.192093e−07, range = [−1.000000, 1.000000]


i= 1, error bound of k( 1) = 2.384186e−07, range = [−1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.384186e−07, range = [−1.000000, 1.000000]

error bound of E_{m} = 4.172325e−07, range = [0.000000, 2.000000]


−−−−−−−−−− Iteration m = 2
Levinson−Durbin algorithm for case [−1, 1]

AA bound = Hard, precision = 24, r in [−1.00, 1.00]
error bound of E_{m−1} = 4.172325e−07, range = [0.000000, 2.000000]
error bound of beta    = 3.576279e−07, range = [−2.000000, 2.000000]

Warning: The input range [0.000000, 2.000000] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1072
  In AA_bound_levinson_wAAFloat_general_range at 88
The user does not specify any range for the input interval of the inverse operation.
AAFloat is using the default range [1.192093e−07, 2.000000e+00] to compute the inverse
    .

i= 1, error bound of k( 1) = 2.384186e−07, range = [−1.000000, 1.000000]
i= 2, error bound of k( 2) = 5.872026e+07, range = [−16777216.000000, 16777216.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 5.872026e+07, range = [−16777217.000000, 16777217.000000]
i= 2, error bound of a( 2) = 5.872026e+07, range = [−16777216.000000, 16777216.000000]

error bound of E_{m} = 1.174405e+08, range = [−33554432.000000, 33554434.000000]


−−−−−−−−−− Iteration m = 3
Levinson−Durbin algorithm for case [−1, 1]

AA bound = Hard, precision = 24, r in [−1.00, 1.00]
error bound of E_{m−1} = 1.174405e+08, range = [−33554432.000000, 33554434.000000]
error bound of beta    = 1.174405e+08, range = [−33554434.000000, 33554434.000000]

Warning: The input range [−33554432.000000, 33554434.000000] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1072
  In AA_bound_levinson_wAAFloat_general_range at 88
The user does not specify any range for the input interval of the inverse operation.
AAFloat is using the default range [1.192093e−07, 3.355443e+07] to compute the inverse
    .
Warning: Possibility of huge rounding error!
> In AAFloat.AAFloat>AAFloat.inv at 1016
  In AAFloat.AAFloat>AAFloat.div at 1072
  In AA_bound_levinson_wAAFloat_general_range at 88
In inverse operation, error component estimation: range of inverse contains ZERO [0.000000e+00, 1.125900e+15].
AAFloat is using the range [1.192093e-07, 1.125900e+15] instead. However, huge rounding error may result!
range = [0.000000e+00, 1.125900e+15]
??? Error using ==> AAFloat.AAFloat>aa_minrange_inv at 1871
Error: in aa_minrange_inv approximation: LOWER BOUND of input affine interval equals ZERO!

Error in ==> AAFloat.AAFloat>AAFloat.inv at 1031
            xsquareinv          = aa_minrange_inv (xsquare_range);

Error in ==> AAFloat.AAFloat>AAFloat.div at 1072
            yy = inv(y, prob);

Error in ==> AA_bound_levinson_wAAFloat_general_range at 88
    kk = -div(LD_beta, LD_alpha, prob);              % beta/alpha
```

E.3 Report for scenario 4

```
*******************************************************************
*******************************************************************

---------- Iteration m = 1
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 0.000000e+00, range = [1.000000, 1.000000]
error bound of beta    = 2.741814e-08, range = [0.170000, 0.230000]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 5.483627e-08, range = [-0.230000, -0.170000]

error bound of E_{m} = 1.302063e-07, range = [0.947100, 0.972900]


---------- Iteration m = 2
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 1.302063e-07, range = [0.947100, 0.972900]
error bound of beta    = 4.581809e-08, range = [0.117100, 0.202900]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.006850e-07, range = [-0.201660, -0.131661]
```

```
i= 2, error bound of a( 2) = 1.017001e-07, range = [-0.210176, -0.123218]

error bound of E_{m} = 2.177255e-07, range = [0.912108, 0.954549]


---------- Iteration m = 3
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 2.177255e-07, range = [0.912108, 0.954549]
error bound of beta    = 8.074333e-08, range = [0.082108, 0.184549]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.456357e-07, range = [-0.192202, -0.093469]
i= 2, error bound of a( 2) = 1.365928e-07, range = [-0.199575, -0.086178]
i= 3, error bound of a( 3) = 1.594273e-07, range = [-0.196811, -0.089042]

error bound of E_{m} = 3.121425e-07, range = [0.885643, 0.942902]


---------- Iteration m = 4
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 3.121425e-07, range = [0.885643, 0.942902]
error bound of beta    = 1.245731e-07, range = [0.055643, 0.172902]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.819867e-07, range = [-0.186580, -0.063328]
i= 2, error bound of a( 2) = 1.817985e-07, range = [-0.183114, -0.066889]
i= 3, error bound of a( 3) = 1.897625e-07, range = [-0.191003, -0.059110]
i= 4, error bound of a( 4) = 2.314679e-07, range = [-0.189101, -0.061118]

error bound of E_{m} = 4.099125e-07, range = [0.863813, 0.936138]


---------- Iteration m = 5
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 4.099125e-07, range = [0.863813, 0.936138]
error bound of beta    = 1.711859e-07, range = [0.033982, 0.165970]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
```

```
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.116300e-07, range = [-0.184987, -0.037079]
i= 2, error bound of a( 2) = 2.218520e-07, range = [-0.179995, -0.042179]
i= 3, error bound of a( 3) = 2.024525e-07, range = [-0.188294, -0.034002]
i= 4, error bound of a( 4) = 2.579072e-07, range = [-0.187487, -0.034927]
i= 5, error bound of a( 5) = 3.109584e-07, range = [-0.185614, -0.036920]

error bound of E_{m} = 5.085796e-07, range = [0.843477, 0.934226]


---------- Iteration m = 6
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 5.085796e-07, range = [0.843477, 0.934226]
error bound of beta    = 2.237670e-07, range = [0.013486, 0.164218]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.371997e-07, range = [-0.187688, -0.012076]
i= 2, error bound of a( 2) = 2.545590e-07, range = [-0.182601, -0.017281]
i= 3, error bound of a( 3) = 2.647884e-07, range = [-0.180427, -0.019590]
i= 4, error bound of a( 4) = 2.776551e-07, range = [-0.189028, -0.011118]
i= 5, error bound of a( 5) = 3.423762e-07, range = [-0.188213, -0.012064]
i= 6, error bound of a( 6) = 4.054436e-07, range = [-0.187718, -0.012729]

error bound of E_{m} = 6.092186e-07, range = [0.822642, 0.937252]


---------- Iteration m = 7
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 6.092186e-07, range = [0.822642, 0.937252]
error bound of beta    = 2.731665e-07, range = [-0.007290, 0.167183]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
```

```

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.616557e-07, range = [-0.195717, 0.014243]
i= 2, error bound of a( 2) = 2.848413e-07, range = [-0.188962, 0.007353]
i= 3, error bound of a( 3) = 3.080051e-07, range = [-0.187698, 0.005942]
i= 4, error bound of a( 4) = 2.935171e-07, range = [-0.194346, 0.012449]
i= 5, error bound of a( 5) = 3.650987e-07, range = [-0.194408, 0.012368]
i= 6, error bound of a( 6) = 4.403941e-07, range = [-0.194990, 0.012770]
i= 7, error bound of a( 7) = 5.059894e-07, range = [-0.195617, 0.013135]

error bound of E_{m} = 7.149309e-07, range = [0.798357, 0.946947]


---------- Iteration m = 8
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 7.149309e-07, range = [0.798357, 0.946947]
error bound of beta    = 3.307788e-07, range = [-0.031643, 0.176947]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 3.012806e-07, range = [-0.211312, 0.045142]
i= 2, error bound of a( 2) = 3.195183e-07, range = [-0.203414, 0.037086]
i= 3, error bound of a( 3) = 3.483065e-07, range = [-0.201084, 0.034596]
i= 4, error bound of a( 4) = 3.810710e-07, range = [-0.196403, 0.029761]
i= 5, error bound of a( 5) = 3.803162e-07, range = [-0.205995, 0.039198]
i= 6, error bound of a( 6) = 4.718850e-07, range = [-0.207282, 0.040292]
i= 7, error bound of a( 7) = 5.589269e-07, range = [-0.208989, 0.041726]
i= 8, error bound of a( 8) = 6.386663e-07, range = [-0.213076, 0.045351]

error bound of E_{m} = 8.339088e-07, range = [0.766725, 0.966394]


---------- Iteration m = 9
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 8.339088e-07, range = [0.766725, 0.966394]
error bound of beta    = 4.001061e-07, range = [-0.063275, 0.196394]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
```

```
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e-07, range = [-0.246109, 0.090425]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 3.841044e-07, range = [-0.239477, 0.086362]
i= 2, error bound of a( 2) = 3.764219e-07, range = [-0.229079, 0.075772]
i= 3, error bound of a( 3) = 3.912121e-07, range = [-0.225934, 0.072444]
i= 4, error bound of a( 4) = 4.489043e-07, range = [-0.219977, 0.066318]
i= 5, error bound of a( 5) = 4.106854e-07, range = [-0.226365, 0.072539]
i= 6, error bound of a( 6) = 4.862096e-07, range = [-0.229658, 0.075628]
i= 7, error bound of a( 7) = 6.073359e-07, range = [-0.232115, 0.077800]
i= 8, error bound of a( 8) = 7.188294e-07, range = [-0.237448, 0.082658]
i= 9, error bound of a( 9) = 8.302704e-07, range = [-0.246109, 0.090425]

error bound of E_{m} = 9.818680e-07, range = [0.718425, 1.004332]


----------- Iteration m = 10
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 9.818680e-07, range = [0.718425, 1.004332]
error bound of beta    = 4.990450e-07, range = [-0.111575, 0.234332]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e-07, range = [-0.246109, 0.090425]
i=10, error bound of k(10) = 1.178539e-06, range = [-0.313863, 0.167315]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 5.677371e-07, range = [-0.293483, 0.151776]
i= 2, error bound of a( 2) = 5.017623e-07, range = [-0.278191, 0.136226]
i= 3, error bound of a( 3) = 4.695504e-07, range = [-0.272332, 0.130149]
i= 4, error bound of a( 4) = 5.318313e-07, range = [-0.264763, 0.122391]
i= 5, error bound of a( 5) = 5.752593e-07, range = [-0.256556, 0.114001]
i= 6, error bound of a( 6) = 5.191886e-07, range = [-0.270415, 0.127644]
i= 7, error bound of a( 7) = 6.537136e-07, range = [-0.275206, 0.132137]
i= 8, error bound of a( 8) = 7.919249e-07, range = [-0.281671, 0.138114]
i= 9, error bound of a( 9) = 9.624451e-07, range = [-0.291951, 0.147487]
i=10, error bound of a(10) = 1.178539e-06, range = [-0.313863, 0.167315]

error bound of E_{m} = 1.200439e-06, range = [0.630733, 1.083029]


----------- Iteration m = 11
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

```

```
AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 1.200439e-06, range = [0.630733, 1.083029]
error bound of beta    = 6.677230e-07, range = [-0.199176, 0.312938]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e-07, range = [-0.246109, 0.090425]
i=10, error bound of k(10) = 1.178539e-06, range = [-0.313863, 0.167315]
i=11, error bound of k(11) = 2.061141e-06, range = [-0.480281, 0.337578]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.003311e-06, range = [-0.420612, 0.289361]
i= 2, error bound of a( 2) = 9.741858e-07, range = [-0.391389, 0.259732]
i= 3, error bound of a( 3) = 8.470351e-07, range = [-0.375696, 0.243756]
i= 4, error bound of a( 4) = 8.254623e-07, range = [-0.367075, 0.234911]
i= 5, error bound of a( 5) = 9.307695e-07, range = [-0.355374, 0.223007]
i= 6, error bound of a( 6) = 8.075530e-07, range = [-0.360570, 0.227970]
i= 7, error bound of a( 7) = 8.740120e-07, range = [-0.370919, 0.238009]
i= 8, error bound of a( 8) = 1.048553e-06, range = [-0.381153, 0.247742]
i= 9, error bound of a( 9) = 1.245666e-06, range = [-0.394048, 0.259712]
i=10, error bound of a(10) = 1.405748e-06, range = [-0.420537, 0.284100]
i=11, error bound of a(11) = 2.061141e-06, range = [-0.480281, 0.337578]

error bound of E_{m} = 2.151582e-06, range = [0.420082, 1.285563]


---------- Iteration m = 12
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 2.151582e-06, range = [0.420082, 1.285563]
error bound of beta    = 1.568348e-06, range = [-0.409854, 0.515499]


i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e-07, range = [-0.246109, 0.090425]
i=10, error bound of k(10) = 1.178539e-06, range = [-0.313863, 0.167315]
i=11, error bound of k(11) = 2.061141e-06, range = [-0.480281, 0.337578]
i=12, error bound of k(12) = 1.016107e-05, range = [-1.206347, 1.039515]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 3.774775e-06, range = [-0.968911, 0.849564]
```

```
i= 2, error bound of a( 2) = 4.973436e-06, range = [-0.860309, 0.740033]
i= 3, error bound of a( 3) = 4.347825e-06, range = [-0.806564, 0.685830]
i= 4, error bound of a( 4) = 3.956194e-06, range = [-0.783025, 0.661990]
i= 5, error bound of a( 5) = 3.808731e-06, range = [-0.757836, 0.636556]
i= 6, error bound of a( 6) = 4.006316e-06, range = [-0.720752, 0.599213]
i= 7, error bound of a( 7) = 3.711993e-06, range = [-0.751849, 0.629980]
i= 8, error bound of a( 8) = 3.734243e-06, range = [-0.777295, 0.654909]
i= 9, error bound of a( 9) = 4.044038e-06, range = [-0.800614, 0.677284]
i=10, error bound of a(10) = 4.512724e-06, range = [-0.844361, 0.718906]
i=11, error bound of a(11) = 3.358012e-06, range = [-0.932835, 0.801081]
i=12, error bound of a(12) = 1.016107e-05, range = [-1.206347, 1.039515]

error bound of E_{m} = 1.149381e-05, range = [-0.546389, 2.243222]


----------- Iteration m = 13
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 1.149381e-05, range = [-0.546389, 2.243222]
error bound of beta    = 1.086233e-05, range = [-1.376382, 1.473215]

Warning: The input range [-0.546389, 2.243222] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1075
  In AA_bound_levinson_wAAFloat_simple_example_increase_order_user at 91
AAFloat is using the specified range [1.000000e-04, 2.243222e+00] to compute the
    inverse.

i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e-07, range = [-0.246109, 0.090425]
i=10, error bound of k(10) = 1.178539e-06, range = [-0.313863, 0.167315]
i=11, error bound of k(11) = 2.061141e-06, range = [-0.480281, 0.337578]
i=12, error bound of k(12) = 1.016107e-05, range = [-1.206347, 1.039515]
i=13, error bound of k(13) = 1.693396e+03, range = [-14732.130514, 14247.947307]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.042676e+03, range = [-17703.636516, 17743.905775]
i= 2, error bound of a( 2) = 1.579637e+03, range = [-13643.732743, 13675.509030]
i= 3, error bound of a( 3) = 1.429796e+03, range = [-12338.766717, 12369.017576]
i= 4, error bound of a( 4) = 1.355713e+03, range = [-11694.501938, 11724.237966]
i= 5, error bound of a( 5) = 1.316224e+03, range = [-11351.470028, 11380.977579]
i= 6, error bound of a( 6) = 1.273133e+03, range = [-10978.563682, 11007.945617]
i= 7, error bound of a( 7) = 1.220472e+03, range = [-10518.377831, 10547.679534]
i= 8, error bound of a( 8) = 1.283268e+03, range = [-11062.862906, 11092.101486]
i= 9, error bound of a( 9) = 1.325925e+03, range = [-11431.582881, 11460.761158]
i=10, error bound of a(10) = 1.365787e+03, range = [-11775.329902, 11804.433132]
i=11, error bound of a(11) = 1.456799e+03, range = [-12559.615033, 12588.601146]
i=12, error bound of a(12) = 1.640724e+03, range = [-14142.905872, 14171.631957]
```

```
i=13, error bound of a(13) = 1.693396e+03, range = [-14732.130514, 14247.947307]

error bound of E_{m} = 4.366231e+03, range = [-33298.047930, 33276.302469]


------------ Iteration m = 14
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 4.366231e+03, range = [-33298.047930, 33276.302469]
error bound of beta    = 4.366231e+03, range = [-33297.193749, 33273.848288]

Warning: The input range [-33298.047930, 33276.302469] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1075
  In AA_bound_levinson_wAAFloat_simple_example_increase_order_user at 91
AAFloat is using the specified range [1.000000e-04, 3.327630e+04] to compute the inverse.
Warning: Possibility of huge rounding error!
> In AAFloat.AAFloat>AAFloat.inv at 1016
  In AAFloat.AAFloat>AAFloat.div at 1075
  In AA_bound_levinson_wAAFloat_simple_example_increase_order_user at 91
In inverse operation, error component estimation: range of inverse contains ZERO [0.000000e+00, 1.107312e+09].
AAFloat is using the range [1.192093e-07, 1.107312e+09] instead. However, huge rounding error may result!

i= 1, error bound of k( 1) = 5.483627e-08, range = [-0.230000, -0.170000]
i= 2, error bound of k( 2) = 1.017001e-07, range = [-0.210176, -0.123218]
i= 3, error bound of k( 3) = 1.594273e-07, range = [-0.196811, -0.089042]
i= 4, error bound of k( 4) = 2.314679e-07, range = [-0.189101, -0.061118]
i= 5, error bound of k( 5) = 3.109584e-07, range = [-0.185614, -0.036920]
i= 6, error bound of k( 6) = 4.054436e-07, range = [-0.187718, -0.012729]
i= 7, error bound of k( 7) = 5.059894e-07, range = [-0.195617, 0.013135]
i= 8, error bound of k( 8) = 6.386663e-07, range = [-0.213076, 0.045351]
i= 9, error bound of k( 9) = 8.302704e-07, range = [-0.246109, 0.090425]
i=10, error bound of k(10) = 1.178539e-06, range = [-0.313863, 0.167315]
i=11, error bound of k(11) = 2.061141e-06, range = [-0.480281, 0.337578]
i=12, error bound of k(12) = 1.016107e-05, range = [-1.206347, 1.039515]
i=13, error bound of k(13) = 1.693396e+03, range = [-14732.130514, 14247.947307]
i=14, error bound of k(14) = 6.097816e+14, range = [-332855218.593768, 332971945.899217]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 8.983381e+18, range = [-4905385163184.478516, 4905356904524.207031]
i= 2, error bound of a( 2) = 8.641599e+18, range = [-4718367274787.789062, 4718368951378.783203]
i= 3, error bound of a( 3) = 7.676296e+18, range = [-4191305525209.088379, 4191307216974.775879]
i= 4, error bound of a( 4) = 7.198125e+18, range = [-3930221399701.955078, 3930223098302.466797]
i= 5, error bound of a( 5) = 6.988560e+18, range = [-3815797726915.217285, 3815799429895.504395]
i= 6, error bound of a( 6) = 6.763759e+18, range = [-3693054691335.196777, 3693056397834.930176]
```

```
i= 7, error bound of a( 7) = 6.431780e+18, range = [-3511792854216.130859,
      3511794564399.884766]
i= 8, error bound of a( 8) = 6.712442e+18, range = [-3665034600931.253906,
      3665036315797.547852]
i= 9, error bound of a( 9) = 6.939910e+18, range = [-3789231520924.349609,
      3789233243121.932617]
i=10, error bound of a(10) = 7.149224e+18, range = [-3903515231874.911621,
      3903516967407.213379]
i=11, error bound of a(11) = 7.542398e+18, range = [-4118184308562.841309,
      4118186074142.467285]
i=12, error bound of a(12) = 8.339073e+18, range = [-4553152579666.305664,
      4553154434275.211914]
i=13, error bound of a(13) = 1.081991e+19, range = [-5907561712856.111328,
      5907564062632.947266]
i=14, error bound of a(14) = 6.097816e+14, range = [-332855218.593768,
      332971945.899217]

error bound of E_{m} = 2.304302e+19, range = [-12557814070480.841797,
      12557812707932.751953]


----------- Iteration m = 15
Levinson-Durbin algorithm for special case [0.17, 0.23], with user-range

AA bound = Hard, precision = 24, r in [ 0.17, 0.23]
error bound of E_{m-1} = 2.304302e+19, range = [-12557814070480.841797,
      12557812707932.751953]
error bound of beta    = 2.304302e+19, range = [-12557814080365.828125,
      12557812717816.136719]

Warning: The input range [-12557814070480.841797, 12557812707932.751953] contains ZERO
      .
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1075
  In AA_bound_levinson_wAAFloat_simple_example_increase_order_user at 91
AAFloat is using the specified range [1.000000e-04, 1.255781e+13] to compute the
      inverse.
Warning: Possibility of huge rounding error!
> In AAFloat.AAFloat>AAFloat.inv at 1016
  In AAFloat.AAFloat>AAFloat.div at 1075
  In AA_bound_levinson_wAAFloat_simple_example_increase_order_user at 91
In inverse operation, error component estimation: range of inverse contains ZERO
      [0.000000e+00, 1.576987e+26].
AAFloat is using the range [1.192093e-07, 1.576987e+26] instead. However, huge
      rounding error may result!
range = [0.000000e+00, 1.576987e+26]
??? Error using ==> AAFloat.AAFloat>aa_minrange_inv at 1875
Error: in aa_minrange_inv approximation: LOWER BOUND of input affine interval equals
      ZERO!

Error in ==> AAFloat.AAFloat>AAFloat.inv at 1031
            xsquareinv          = aa_minrange_inv (xsquare_range);

Error in ==> AAFloat.AAFloat>AAFloat.div at 1075
            yy = inv(y, prob, user_range);

```

```
Error in ==> AA_bound_levinson_wAAFloat_simple_example_increase_order_user at 91
    kk = -div(LD_beta, LD_alpha, prob, [user_lo_bound, user_hi_bound]);           %
         beta/alpha
```

E.4 Report for scenario 5

```
*********************************************************************
*********************************************************************

---------- Iteration m = 1
Levinson-Durbin algorithm for case [-1, 1], with user-range

AA bound = Hard, precision = 24, r in [ 1.00,-1.00]
error bound of E_{m-1} = 0.000000e+00, range = [1.000000, 1.000000]
error bound of beta    = 1.192093e-07, range = [-1.000000, 1.000000]


i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.384186e-07, range = [-1.000000, 1.000000]

error bound of E_{m} = 4.172325e-07, range = [0.000000, 2.000000]


---------- Iteration m = 2
Levinson-Durbin algorithm for case [-1, 1], with user-range

AA bound = Hard, precision = 24, r in [ 1.00,-1.00]
error bound of E_{m-1} = 4.172325e-07, range = [0.000000, 2.000000]
error bound of beta    = 3.576279e-07, range = [-2.000000, 2.000000]

Warning: The input range [0.000000, 2.000000] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1075
  In AA_bound_levinson_wAAFloat_general_range_user at 91
AAFloat is using the specified range [1.000000e-04, 2.000000e+00] to compute the
    inverse.

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 8.345246e+01, range = [-20000.000000, 20000.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 8.345246e+01, range = [-20001.000000, 20001.000000]
i= 2, error bound of a( 2) = 8.345246e+01, range = [-20000.000000, 20000.000000]

error bound of E_{m} = 1.669109e+02, range = [-40000.000000, 40002.000000]


---------- Iteration m = 3
Levinson-Durbin algorithm for case [-1, 1], with user-range

```

```
AA bound = Hard, precision = 24, r in [ 1.00,-1.00]
error bound of E_{m-1} = 1.669109e+02, range = [-40000.000000, 40002.000000]
error bound of beta    = 1.669121e+02, range = [-40002.000000, 40002.000000]

Warning: The input range [-40000.000000, 40002.000000] contains ZERO.
> In AAFloat.AAFloat>AAFloat.inv at 918
  In AAFloat.AAFloat>AAFloat.div at 1075
  In AA_bound_levinson_wAAFloat_general_range_user at 91
AAFloat is using the specified range [1.000000e-04, 4.000200e+04] to compute the
    inverse.
Warning: Possibility of huge rounding error!
> In AAFloat.AAFloat>AAFloat.inv at 1016
  In AAFloat.AAFloat>AAFloat.div at 1075
  In AA_bound_levinson_wAAFloat_general_range_user at 91
In inverse operation, error component estimation: range of inverse contains ZERO
    [0.000000e+00, 1.600160e+09].
AAFloat is using the range [1.192093e-07, 1.600160e+09] instead. However, huge
    rounding error may result!
range = [0.000000e+00, 1.600160e+09]
??? Error using ==> AAFloat.AAFloat>aa_minrange_inv at 1875
Error: in aa_minrange_inv approximation: LOWER BOUND of input affine interval equals
    ZERO!

Error in ==> AAFloat.AAFloat>AAFloat.inv at 1031
            xsquareinv          = aa_minrange_inv (xsquare_range);

Error in ==> AAFloat.AAFloat>AAFloat.div at 1075
            yy = inv(y, prob, user_range);

Error in ==> AA_bound_levinson_wAAFloat_general_range_user at 91
    kk = -div(LD_beta, LD_alpha, prob, [user_lo_bound, user_hi_bound]);             %
        beta/alpha
```

E.5 Report for case using the AA-based scaling operator (AASO): Hard error bound

```
---------- Iteration m = 1
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Hard, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 0.000000e+00, range = [1.000000, 1.000000]

Before performing AASO:
Variable "beta": range = [-1.000000, 1.000000]; error bound = 1.192093e-07
AASO: specified range [-1.000000,1.000000] is larger than or equal to original range
    [-1.000000,1.000000]. No scaling performed!
error bound of beta = 1.192093e-07, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-1.000000, 1.000000]; error bound = 2.384186e-07
```

```
AASO: specified range [-1.000000,1.000000] is larger than or equal to original range
    [-1.000000,1.000000]. No scaling performed!

Before performing AASO:
Variable "E_{m-1}": range = [0.000000, 1.000000]; error bound = 6.556511e-07
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.384186e-07, range = [-1.000000, 1.000000]

error bound of E_{m} = 6.555855e-07, range = [0.000100, 1.000000]


----------- Iteration m = 2
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Hard, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 6.555855e-07, range = [0.000100, 1.000000]

Before performing AASO:
Variable "beta": range = [-2.000000, 2.000000]; error bound = 3.576279e-07
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 1.788139e-07, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-10000.000000, 10000.000000]; error bound = 6.556034e+01
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.499950, 1.000000]; error bound = 1.311290e-02
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 6.556034e-03, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 6.556690e-03, range = [-2.000000, 2.000000]
i= 2, error bound of a( 2) = 6.556034e-03, range = [-1.000000, 1.000000]

error bound of E_{m} = 8.741352e-03, range = [0.000100, 1.000000]


----------- Iteration m = 3
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Hard, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 8.741352e-03, range = [0.000100, 1.000000]

Before performing AASO:
Variable "beta": range = [-4.000000, 4.000000]; error bound = 1.311314e-02
AASO: perform range scaling with specified range [-1.000000,1.000000]
```

```
error bound of beta = 3.278285e−03, range = [−1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [−10000.000000, 10000.000000]; error bound = 8.741025e+05
AASO: perform range scaling with specified range [−1.000000,1.000000]

Before performing AASO:
Variable "E_{m−1}": range = [−0.499950, 1.000000]; error bound = 1.748292e+02
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e−07, range = [−1.000000, 1.000000]
i= 2, error bound of k( 2) = 6.556034e−03, range = [−1.000000, 1.000000]
i= 3, error bound of k( 3) = 8.741025e+01, range = [−1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 8.742336e+01, range = [−3.000000, 3.000000]
i= 2, error bound of a( 2) = 1.748336e+02, range = [−3.000000, 3.000000]
i= 3, error bound of a( 3) = 8.741025e+01, range = [−1.000000, 1.000000]

error bound of E_{m} = 1.165451e+02, range = [0.000100, 1.000000]


−−−−−−−−−−− Iteration m = 4
Levinson−Durbin algorithm for general range [−1,+1] using range−scaling

AA bound = Hard, precision = 24, r in [−1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m−1} = 1.165451e+02, range = [0.000100, 1.000000]

Before performing AASO:
Variable "beta": range = [−8.000000, 8.000000]; error bound = 3.496672e+02
AASO: perform range scaling with specified range [−1.000000,1.000000]
error bound of beta = 4.370840e+01, range = [−1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [−10000.000000, 10000.000000]; error bound = 1.165407e+10
AASO: perform range scaling with specified range [−1.000000,1.000000]

Before performing AASO:
Variable "E_{m−1}": range = [−0.499950, 1.000000]; error bound = 2.330930e+06
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e−07, range = [−1.000000, 1.000000]
i= 2, error bound of k( 2) = 6.556034e−03, range = [−1.000000, 1.000000]
i= 3, error bound of k( 3) = 8.741025e+01, range = [−1.000000, 1.000000]
i= 4, error bound of k( 4) = 1.165407e+06, range = [−1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.165582e+06, range = [−4.000000, 4.000000]
i= 2, error bound of a( 2) = 3.496570e+06, range = [−6.000000, 6.000000]
i= 3, error bound of a( 3) = 3.496395e+06, range = [−4.000000, 4.000000]
i= 4, error bound of a( 4) = 1.165407e+06, range = [−1.000000, 1.000000]

error bound of E_{m} = 1.553850e+06, range = [0.000100, 1.000000]
```

```


---------- Iteration m = 5
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Hard, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 1.553850e+06, range = [0.000100, 1.000000]

Before performing AASO:
Variable "beta": range = [-16.000000, 16.000000]; error bound = 9.323954e+06
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 5.827471e+05, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-10000.000000, 10000.000000]; error bound = 1.553792e+14
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.499950, 1.000000]; error bound = 3.107739e+10
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 6.556034e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 8.741025e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 1.165407e+06, range = [-1.000000, 1.000000]
i= 5, error bound of k( 5) = 1.553792e+10, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.554025e+10, range = [-5.000000, 5.000000]
i= 2, error bound of a( 2) = 6.215866e+10, range = [-10.000000, 10.000000]
i= 3, error bound of a( 3) = 9.323449e+10, range = [-10.000000, 10.000000]
i= 4, error bound of a( 4) = 6.215399e+10, range = [-5.000000, 5.000000]
i= 5, error bound of a( 5) = 1.553792e+10, range = [-1.000000, 1.000000]

error bound of E_{m} = 2.071688e+10, range = [0.000100, 1.000000]


---------- Iteration m = 6
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Hard, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 2.071688e+10, range = [0.000100, 1.000000]

Before performing AASO:
Variable "beta": range = [-32.000000, 32.000000]; error bound = 2.486253e+11
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 7.769541e+09, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-10000.000000, 10000.000000]; error bound = 2.071610e+18
AASO: perform range scaling with specified range [-1.000000,1.000000]

```

```
Before performing AASO:
Variable "E_{m-1}": range = [-0.499950, 1.000000]; error bound = 4.143427e+14
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 6.556034e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 8.741025e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 1.165407e+06, range = [-1.000000, 1.000000]
i= 5, error bound of k( 5) = 1.553792e+10, range = [-1.000000, 1.000000]
i= 6, error bound of k( 6) = 2.071610e+14, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.071921e+14, range = [-6.000000, 6.000000]
i= 2, error bound of a( 2) = 1.035929e+15, range = [-15.000000, 15.000000]
i= 3, error bound of a( 3) = 2.071796e+15, range = [-20.000000, 20.000000]
i= 4, error bound of a( 4) = 2.071734e+15, range = [-15.000000, 15.000000]
i= 5, error bound of a( 5) = 1.035836e+15, range = [-6.000000, 6.000000]
i= 6, error bound of a( 6) = 2.071610e+14, range = [-1.000000, 1.000000]

error bound of E_{m} = 2.762100e+14, range = [0.000100, 1.000000]


---------- Iteration m = 7
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Hard, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 2.762100e+14, range = [0.000100, 1.000000]

Before performing AASO:
Variable "beta": range = [-64.000000, 64.000000]; error bound = 6.629649e+15
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 1.035883e+14, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-10000.000000, 10000.000000]; error bound = 2.761997e+22
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.499950, 1.000000]; error bound = 5.524270e+18
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 6.556034e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 8.741025e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 1.165407e+06, range = [-1.000000, 1.000000]
i= 5, error bound of k( 5) = 1.553792e+10, range = [-1.000000, 1.000000]
i= 6, error bound of k( 6) = 2.071610e+14, range = [-1.000000, 1.000000]
i= 7, error bound of k( 7) = 2.761997e+18, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.762411e+18, range = [-7.000000, 7.000000]
i= 2, error bound of a( 2) = 1.657405e+19, range = [-21.000000, 21.000000]
i= 3, error bound of a( 3) = 4.143410e+19, range = [-35.000000, 35.000000]
i= 4, error bound of a( 4) = 5.524408e+19, range = [-35.000000, 35.000000]
```

```
i= 5, error bound of a( 5) = 4.143203e+19, range = [-21.000000, 21.000000]
i= 6, error bound of a( 6) = 1.657240e+19, range = [-7.000000, 7.000000]
i= 7, error bound of a( 7) = 2.761997e+18, range = [-1.000000, 1.000000]

error bound of E_{m} = 3.682601e+18, range = [0.000100, 1.000000]


------------ Iteration m = 8
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Hard, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 3.682601e+18, range = [0.000100, 1.000000]

Before performing AASO:
Variable "beta": range = [-128.000000, 128.000000]; error bound = 1.767811e+20
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 1.381102e+18, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-10000.000000, 10000.000000]; error bound = 3.682463e+26
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.499950, 1.000000]; error bound = 7.365294e+22
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 6.556034e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 8.741025e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 1.165407e+06, range = [-1.000000, 1.000000]
i= 5, error bound of k( 5) = 1.553792e+10, range = [-1.000000, 1.000000]
i= 6, error bound of k( 6) = 2.071610e+14, range = [-1.000000, 1.000000]
i= 7, error bound of k( 7) = 2.761997e+18, range = [-1.000000, 1.000000]
i= 8, error bound of k( 8) = 3.682463e+22, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 3.683015e+22, range = [-8.000000, 8.000000]
i= 2, error bound of a( 2) = 2.578056e+23, range = [-28.000000, 28.000000]
i= 3, error bound of a( 3) = 7.734001e+23, range = [-56.000000, 56.000000]
i= 4, error bound of a( 4) = 1.288973e+24, range = [-70.000000, 70.000000]
i= 5, error bound of a( 5) = 1.288945e+24, range = [-56.000000, 56.000000]
i= 6, error bound of a( 6) = 7.733504e+23, range = [-28.000000, 28.000000]
i= 7, error bound of a( 7) = 2.577779e+23, range = [-8.000000, 8.000000]
i= 8, error bound of a( 8) = 3.682463e+22, range = [-1.000000, 1.000000]

error bound of E_{m} = 4.909869e+22, range = [0.000100, 1.000000]


------------ Iteration m = 9
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Hard, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 4.909869e+22, range = [0.000100, 1.000000]
```

```

Before performing AASO:
Variable "beta": range = [-256.000000, 256.000000]; error bound = 4.713906e+24
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 1.841370e+22, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-10000.000000, 10000.000000]; error bound = 4.909685e+30
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.499950, 1.000000]; error bound = 9.819861e+26
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 6.556034e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 8.741025e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 1.165407e+06, range = [-1.000000, 1.000000]
i= 5, error bound of k( 5) = 1.553792e+10, range = [-1.000000, 1.000000]
i= 6, error bound of k( 6) = 2.071610e+14, range = [-1.000000, 1.000000]
i= 7, error bound of k( 7) = 2.761997e+18, range = [-1.000000, 1.000000]
i= 8, error bound of k( 8) = 3.682463e+22, range = [-1.000000, 1.000000]
i= 9, error bound of k( 9) = 4.909685e+26, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 4.910421e+26, range = [-9.000000, 9.000000]
i= 2, error bound of a( 2) = 3.928263e+27, range = [-36.000000, 36.000000]
i= 3, error bound of a( 3) = 1.374866e+28, range = [-84.000000, 84.000000]
i= 4, error bound of a( 4) = 2.749681e+28, range = [-126.000000, 126.000000]
i= 5, error bound of a( 5) = 3.437037e+28, range = [-126.000000, 126.000000]
i= 6, error bound of a( 6) = 2.749578e+28, range = [-84.000000, 84.000000]
i= 7, error bound of a( 7) = 1.374763e+28, range = [-36.000000, 36.000000]
i= 8, error bound of a( 8) = 3.927821e+27, range = [-9.000000, 9.000000]
i= 9, error bound of a( 9) = 4.909685e+26, range = [-1.000000, 1.000000]

error bound of E_{m} = 6.546137e+26, range = [0.000100, 1.000000]


------------ Iteration m = 10
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Hard, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 6.546137e+26, range = [0.000100, 1.000000]

Before performing AASO:
Variable "beta": range = [-512.000000, 512.000000]; error bound = 1.256974e+29
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 2.455027e+26, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-10000.000000, 10000.000000]; error bound = 6.545892e+34
AASO: perform range scaling with specified range [-1.000000,1.000000]

```

```
Before performing AASO:
Variable "E_{m-1}": range = [-0.499950, 1.000000]; error bound = 1.309244e+31
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 6.556034e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 8.741025e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 1.165407e+06, range = [-1.000000, 1.000000]
i= 5, error bound of k( 5) = 1.553792e+10, range = [-1.000000, 1.000000]
i= 6, error bound of k( 6) = 2.071610e+14, range = [-1.000000, 1.000000]
i= 7, error bound of k( 7) = 2.761997e+18, range = [-1.000000, 1.000000]
i= 8, error bound of k( 8) = 3.682463e+22, range = [-1.000000, 1.000000]
i= 9, error bound of k( 9) = 4.909685e+26, range = [-1.000000, 1.000000]
i=10, error bound of k(10) = 6.545892e+30, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 6.546874e+30, range = [-10.000000, 10.000000]
i= 2, error bound of a( 2) = 5.892088e+31, range = [-45.000000, 45.000000]
i= 3, error bound of a( 3) = 2.356796e+32, range = [-120.000000, 120.000000]
i= 4, error bound of a( 4) = 5.499099e+32, range = [-210.000000, 210.000000]
i= 5, error bound of a( 5) = 8.248511e+32, range = [-252.000000, 252.000000]
i= 6, error bound of a( 6) = 8.248374e+32, range = [-210.000000, 210.000000]
i= 7, error bound of a( 7) = 5.498824e+32, range = [-120.000000, 120.000000]
i= 8, error bound of a( 8) = 2.356600e+32, range = [-45.000000, 45.000000]
i= 9, error bound of a( 9) = 5.891401e+31, range = [-10.000000, 10.000000]
i=10, error bound of a(10) = 6.545892e+30, range = [-1.000000, 1.000000]

error bound of E_{m} = 8.727710e+30, range = [0.000100, 1.000000]

Time = 0.349157 (minutes)
```

E.6 Report for case using the AA-based scaling operator (AASO): Probabilistic error bound

```
---------- Iteration m = 1
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 0.000000e+00, range = [1.000000, 1.000000]

Before performing AASO:
Variable "beta": range = [-1.000000, 1.000000]; error bound = 1.192093e-07
AASO: specified range [-1.000000,1.000000] is larger than or equal to original range
    [-1.000000,1.000000]. No scaling performed!
error bound of beta = 1.192093e-07, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-1.000000, 1.000000]; error bound = 2.384186e-07
AASO: specified range [-1.000000,1.000000] is larger than or equal to original range
    [-1.000000,1.000000]. No scaling performed!
```

```

Before performing AASO:
Variable "E_{m-1}": range = [0.000000, 1.000000]; error bound = 2.467309e-07
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.384186e-07, range = [-1.000000, 1.000000]

error bound of E_{m} = 2.467062e-07, range = [0.000100, 1.000000]


----------- Iteration m = 2
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 2.467062e-07, range = [0.000100, 1.000000]

Before performing AASO:
Variable "beta": range = [-2.000000, 2.000000]; error bound = 1.601001e-07
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 8.005007e-08, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-10000.000000, 10000.000000]; error bound = 2.467036e+01
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.115828, 0.615878]; error bound = 4.934319e-03
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 2.467036e-03, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 2.467244e-03, range = [-2.000000, 2.000000]
i= 2, error bound of a( 2) = 2.467036e-03, range = [-1.000000, 1.000000]

error bound of E_{m} = 3.289327e-03, range = [0.256164, 0.743936]


----------- Iteration m = 3
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 3.289327e-03, range = [0.256164, 0.743936]

Before performing AASO:
Variable "beta": range = [-4.000000, 4.000000]; error bound = 4.934176e-03
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 1.233544e-03, range = [-1.000000, 1.000000]

```

```

Before performing AASO:
Variable "k": range = [-10000.000000, 10000.000000]; error bound = 3.206409e+05
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.050977, 0.551027]; error bound = 4.770980e+01
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 2.467036e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 3.206409e+01, range = [-1.000000, 1.000000]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 3.206902e+01, range = [-3.000000, 3.000000]
i= 2, error bound of a( 2) = 6.413311e+01, range = [-3.000000, 3.000000]
i= 3, error bound of a( 3) = 3.206409e+01, range = [-1.000000, 1.000000]

error bound of E_{m} = 3.180441e+01, range = [0.299396, 0.700704]


---------- Iteration m = 4
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 3.180441e+01, range = [0.299396, 0.700704]

Before performing AASO:
Variable "beta": range = [-8.000000, 8.000000]; error bound = 1.282662e+02
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 1.603328e+01, range = [-1.000000, 1.000000]


Before performing AASO:
Variable "k": range = [-5439.557126, 5439.557126]; error bound = 3.100228e+09
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.041379, 0.541429]; error bound = 2.363563e+05
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 2.467036e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 3.206409e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 3.100228e+05, range = [-0.543956, 0.543956]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 3.100723e+05, range = [-4.000000, 4.000000]
i= 2, error bound of a( 2) = 9.301673e+05, range = [-3.473067, 3.473067]
i= 3, error bound of a( 3) = 9.301178e+05, range = [-2.902863, 2.902863]
i= 4, error bound of a( 4) = 3.100228e+05, range = [-0.543956, 0.543956]

error bound of E_{m} = 1.575603e+05, range = [0.305793, 0.694307]


```

```
---------- Iteration m = 5
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 1.575603e+05, range = [0.305793, 0.694307]

Before performing AASO:
Variable "beta": range = [-7.945398, 7.945398]; error bound = 2.480380e+06
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 1.550238e+05, range = [-0.496587, 0.496587]


Before performing AASO:
Variable "k": range = [-5357.974880, 5357.974880]; error bound = 7.625787e+12
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.039400, 0.539450]; error bound = 5.674913e+08
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 2.467036e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 3.206409e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 3.100228e+05, range = [-0.543956, 0.543956]
i= 5, error bound of k( 5) = 7.625787e+08, range = [-0.535797, 0.535797]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 4.152852e+08, range = [-2.123497, 2.123497]
i= 2, error bound of a( 2) = 2.215090e+09, range = [-5.147014, 5.147014]
i= 3, error bound of a( 3) = 2.649915e+09, range = [-6.394774, 6.394774]
i= 4, error bound of a( 4) = 3.050791e+09, range = [-3.837376, 3.837376]
i= 5, error bound of a( 5) = 7.625787e+08, range = [-0.535797, 0.535797]

error bound of E_{m} = 3.783023e+08, range = [0.307113, 0.692987]


---------- Iteration m = 6
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 3.783023e+08, range = [0.307113, 0.692987]

Before performing AASO:
Variable "beta": range = [-15.075333, 15.075333]; error bound = 9.093660e+09
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 2.841769e+08, range = [-0.471104, 0.471104]


Before performing AASO:
Variable "k": range = [-5300.120185, 5300.120185]; error bound = 1.737060e+16
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.038455, 0.538505]; error bound = 1.276304e+12
```

```
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 2.467036e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 3.206409e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 3.100228e+05, range = [-0.543956, 0.543956]
i= 5, error bound of k( 5) = 7.625787e+08, range = [-0.535797, 0.535797]
i= 6, error bound of k( 6) = 1.737060e+12, range = [-0.530012, 0.530012]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 9.315318e+11, range = [-2.326174, 2.326174]
i= 2, error bound of a( 2) = 6.669584e+12, range = [-7.002710, 7.002710]
i= 3, error bound of a( 3) = 1.111216e+13, range = [-11.448926, 11.448926]
i= 4, error bound of a( 4) = 8.944897e+12, range = [-10.242570, 10.242570]
i= 5, error bound of a( 5) = 3.689624e+12, range = [-4.778418, 4.778418]
i= 6, error bound of a( 6) = 1.737060e+12, range = [-0.530012, 0.530012]

error bound of E_{m} = 8.508126e+11, range = [0.307743, 0.692357]


---------- Iteration m = 7
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 8.508126e+11, range = [0.307743, 0.692357]

Before performing AASO:
Variable "beta": range = [-28.867064, 28.867064]; error bound = 3.308486e+13
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 5.169509e+11, range = [-0.451048, 0.451048]


Before performing AASO:
Variable "k": range = [-5256.311287, 5256.311287]; error bound = 3.740470e+19
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.037849, 0.537899]; error bound = 2.723148e+15
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 2.467036e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 3.206409e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 3.100228e+05, range = [-0.543956, 0.543956]
i= 5, error bound of k( 5) = 7.625787e+08, range = [-0.535797, 0.535797]
i= 6, error bound of k( 6) = 1.737060e+12, range = [-0.530012, 0.530012]
i= 7, error bound of k( 7) = 3.740470e+15, range = [-0.525631, 0.525631]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.984339e+15, range = [-2.512555, 2.512555]
i= 2, error bound of a( 2) = 1.788214e+16, range = [-9.027983, 9.027983]
i= 3, error bound of a( 3) = 3.832784e+16, range = [-18.275494, 18.275494]
i= 4, error bound of a( 4) = 4.283915e+16, range = [-21.578910, 21.578910]
i= 5, error bound of a( 5) = 2.620063e+16, range = [-15.024947, 15.024947]
i= 6, error bound of a( 6) = 8.703212e+15, range = [-5.722537, 5.722537]
```

```
i= 7, error bound of a( 7) = 3.740470e+15, range = [-0.525631, 0.525631]

error bound of E_{m} = 1.815311e+15, range = [0.308147, 0.691953]


------------ Iteration m = 8
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 1.815311e+15, range = [0.308147, 0.691953]

Before performing AASO:
Variable "beta": range = [-55.633998, 55.633998]; error bound = 1.396778e+17
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 1.091233e+15, range = [-0.434641, 0.434641]


Before performing AASO:
Variable "k": range = [-5221.632168, 5221.632168]; error bound = 7.690407e+22
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.037386, 0.537436]; error bound = 5.558660e+18
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 2.467036e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 3.206409e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 3.100228e+05, range = [-0.543956, 0.543956]
i= 5, error bound of k( 5) = 7.625787e+08, range = [-0.535797, 0.535797]
i= 6, error bound of k( 6) = 1.737060e+12, range = [-0.530012, 0.530012]
i= 7, error bound of k( 7) = 3.740470e+15, range = [-0.525631, 0.525631]
i= 8, error bound of k( 8) = 7.690407e+18, range = [-0.522163, 0.522163]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 4.046255e+18, range = [-2.686035, 2.686035]
i= 2, error bound of a( 2) = 4.403107e+19, range = [-11.211379, 11.211379]
i= 3, error bound of a( 3) = 1.156000e+20, range = [-27.050112, 27.050112]
i= 4, error bound of a( 4) = 1.660158e+20, range = [-39.628436, 39.628436]
i= 5, error bound of a( 5) = 1.405922e+20, range = [-36.476192, 36.476192]
i= 6, error bound of a( 6) = 6.944690e+19, range = [-20.747586, 20.747586]
i= 7, error bound of a( 7) = 1.932735e+19, range = [-6.668345, 6.668345]
i= 8, error bound of a( 8) = 7.690407e+18, range = [-0.522163, 0.522163]

error bound of E_{m} = 3.705526e+18, range = [0.308455, 0.691645]


------------ Iteration m = 9
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 3.705526e+18, range = [0.308455, 0.691645]

Before performing AASO:
```

```
Variable "beta": range = [-107.734773, 107.734773]; error bound = 5.667500e+20
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 2.213867e+18, range = [-0.420839, 0.420839]


Before performing AASO:
Variable "k": range = [-5193.284870, 5193.284870]; error bound = 1.519961e+26
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.037014, 0.537064]; error bound = 1.092193e+22
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 2.467036e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 3.206409e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 3.100228e+05, range = [-0.543956, 0.543956]
i= 5, error bound of k( 5) = 7.625787e+08, range = [-0.535797, 0.535797]
i= 6, error bound of k( 6) = 1.737060e+12, range = [-0.530012, 0.530012]
i= 7, error bound of k( 7) = 3.740470e+15, range = [-0.525631, 0.525631]
i= 8, error bound of k( 8) = 7.690407e+18, range = [-0.522163, 0.522163]
i= 9, error bound of k( 9) = 1.519961e+22, range = [-0.519328, 0.519328]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 7.944715e+21, range = [-2.848970, 2.848970]
i= 2, error bound of a( 2) = 1.014103e+23, range = [-13.543016, 13.543016]
i= 3, error bound of a( 3) = 3.155068e+23, range = [-37.930954, 37.930954]
i= 4, error bound of a( 4) = 5.546628e+23, range = [-66.322000, 66.322000]
i= 5, error bound of a( 5) = 6.025634e+23, range = [-75.825899, 75.825899]
i= 6, error bound of a( 6) = 4.112805e+23, range = [-57.084648, 57.084648]
i= 7, error bound of a( 7) = 1.704507e+23, range = [-27.413783, 27.413783]
i= 8, error bound of a( 8) = 4.083646e+22, range = [-7.615177, 7.615177]
i= 9, error bound of a( 9) = 1.519961e+22, range = [-0.519328, 0.519328]

error bound of E_{m} = 7.280803e+21, range = [0.308704, 0.691396]


---------- Iteration m = 10
Levinson-Durbin algorithm for general range [-1,+1] using range-scaling

AA bound = Soft, precision = 24, r in [-1.00, 1.00]
delta_E  = 0.000100
error bound of E_{m-1} = 7.280803e+21, range = [0.308704, 0.691396]

Before performing AASO:
Variable "beta": range = [-209.398761, 209.398761]; error bound = 2.219855e+24
AASO: perform range scaling with specified range [-1.000000,1.000000]
error bound of beta = 4.335655e+21, range = [-0.408982, 0.408982]


Before performing AASO:
Variable "k": range = [-5169.543382, 5169.543382]; error bound = 2.902340e+29
AASO: perform range scaling with specified range [-1.000000,1.000000]

Before performing AASO:
Variable "E_{m-1}": range = [-0.036704, 0.536754]; error bound = 2.075263e+25
```

```
AASO: perform range scaling with specified range [0.000100,1.000000]

i= 1, error bound of k( 1) = 2.384186e-07, range = [-1.000000, 1.000000]
i= 2, error bound of k( 2) = 2.467036e-03, range = [-1.000000, 1.000000]
i= 3, error bound of k( 3) = 3.206409e+01, range = [-1.000000, 1.000000]
i= 4, error bound of k( 4) = 3.100228e+05, range = [-0.543956, 0.543956]
i= 5, error bound of k( 5) = 7.625787e+08, range = [-0.535797, 0.535797]
i= 6, error bound of k( 6) = 1.737060e+12, range = [-0.530012, 0.530012]
i= 7, error bound of k( 7) = 3.740470e+15, range = [-0.525631, 0.525631]
i= 8, error bound of k( 8) = 7.690407e+18, range = [-0.522163, 0.522163]
i= 9, error bound of k( 9) = 1.519961e+22, range = [-0.519328, 0.519328]
i=10, error bound of k(10) = 2.902340e+25, range = [-0.516954, 0.516954]

i= 0, error bound of a( 0) = 0.000000e+00, range = [1.000000, 1.000000]
i= 1, error bound of a( 1) = 1.508848e+25, range = [-3.003078, 3.003078]
i= 2, error bound of a( 2) = 2.211409e+26, range = [-16.014461, 16.014461]
i= 3, error bound of a( 3) = 7.960449e+26, range = [-51.064195, 51.064195]
i= 4, error bound of a( 4) = 1.657558e+27, range = [-103.740281, 103.740281]
i= 5, error bound of a( 5) = 2.201640e+27, range = [-141.659108, 141.659108]
i= 6, error bound of a( 6) = 1.925588e+27, range = [-132.575990, 132.575990]
i= 7, error bound of a( 7) = 1.101219e+27, range = [-84.350185, 84.350185]
i= 8, error bound of a( 8) = 3.931577e+26, range = [-35.025501, 35.025501]
i= 9, error bound of a( 9) = 8.270611e+25, range = [-8.562673, 8.562673]
i=10, error bound of a(10) = 2.902340e+25, range = [-0.516954, 0.516954]

error bound of E_{m} = 1.383416e+25, range = [0.308910, 0.691190]

Time = 0.027380 (minutes)
```

Bibliography

[1] J. Kontro, K. Kalliojarvi, and Y. Neuvo, "Floating-point arithmetic in signal processing," in *1992 IEEE International Symposium on Circuits and Systems, 1992 (ISCAS '92)*, vol. 4. IEEE, May 1992, pp. 1784–1791 vol.4. [Online]. Available: http://dx.doi.org/10.1109/ISCAS.1992.230408

[2] A. Lacroix, "Floating-point signal processing-arithmetic, roundoff-noise, and limit cycles," in *IEEE International Symposium on Circuits and Systems, 1988.* IEEE, June 1988, pp. 2023–2030 vol.3. [Online]. Available: http://dx.doi.org/10.1109/ISCAS.1988.15339

[3] W. Krämer, "A priori worst case error bounds for floating-point computations," *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 750–756, July 1998.

[4] C. F. Fang, "Probabilistic Interval-Valued Computation: Representing and Reasoning about Uncertainty in DSP and VLSI Design," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[5] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[6] P. Strobach, *Linear Prediction Theory: A Mathematical Basis for Adaptive Systems*, M. Schroeder, Ed. New York, USA: Springer Series in Information Sciences, Feb. 1990.

[7] P. Vary and R. Martin, *Digital Speech Transmission: Enhancement, Coding, and Error Concealment.* West Sussex PO19 8SQ, England: John Wiley & Sons, LTD, 2006.

[8] S. Haykin, *Adaptive Filter Theory*, 4th ed., T. Kailath, Ed. Upper Saddle River, New Jersey, USA: Prentice Hall Information and System Sciences Series, 2002.

[9] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005. [Online]. Available: http://dx.doi.org/10.1049/ip-cdt:20045086

[10] H. P. Huynh and T. Mitra, "Runtime Adaptive Extensible Embedded Processors – A Survey," in *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. SAMOS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 215–225. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03138-0_23

[11] M. Sima, S. Vassiliadis, S. Cotofana, J. T. J. Eijndhoven, and K. Vissers, "Field-Programmable Custom Computing Machines - A Taxonomy," vol. 2438, pp. 79–88, Aug. 2002. [Online]. Available: http://dx.doi.org/10.1007/3-540-46117-5_10

[12] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in Heterogeneous Computing," *Sci. Program.*, vol. 18, pp. 1–33, Jan. 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1804800

[13] M. Mücke, B. Lesser, and W. Gansterer, "Peak Performance Model for a Custom Precision Floating-Point Dot Product on FPGAs," in *Euro-Par 2010 Parallel Processing Workshops*, ser. Lecture Notes in Computer Science vol. 6586. Italy: Springer, 2011, pp. 399–406.

[14] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, “The MOLEN polymorphic processor,” *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, Nov. 2004. [Online]. Available: http://dx.doi.org/10.1109/TC.2004.104

[15] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, “The GARP architecture and C compiler,” *Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000. [Online]. Available: http://dx.doi.org/10.1109/2.839323

[16] R. E. Gonzalez, “A software-configurable processor architecture,” *IEEE Micro*, vol. 26, no. 5, pp. 42–51, September 2006. [Online]. Available: http://dx.doi.org/10.1109/MM.2006.85

[17] H. P. Huynh, “Instruction-Set Customization for Multi-Tasking Embedded Systems,” Ph.D. dissertation, National University of Singapore, Singapore, 2009.

[18] K.-I. Kum and W. Sung, “Combined word-length optimization and high-level synthesis of digital signal processing systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 921–930, Aug. 2001. [Online]. Available: http://dx.doi.org/10.1109/43.936374

[19] A. A. Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, and N. Shirazi, “Floating-point bitwidth analysis via automatic differentiation,” in *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT).* IEEE, 2002, pp. 158–165. [Online]. Available: http://dx.doi.org/10.1109/FPT.2002.1188677

[20] A. A. Gaffar, O. Mencer, W. Luk, and P. Y. K. Cheung, “Unifying Bit-Width optimisation for Fixed-Point and Floating-Point designs,” in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* IEEE, 2004, pp. 79–88. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2004.59

[21] G. A. Constantinides, “Perturbation analysis for word-length optimization,” in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.* IEEE Comput. Soc, 2003, pp. 81–90. [Online]. Available: http://dx.doi.org/10.1109/FPGA.2003.1227244

[22] B. Lesser, M. Mücke, and W. N. Gansterer, “Effects of reduced precision on floating-point SVM classification accuracy,” in *International Conference on Computational Science (ICCS 2011).* Elsevier, June 2011.

[23] C. F. Fang, T. Chen, and R. A. Rutenbar, “Floating-point error analysis based on affine arithmetic,” in *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP ’03). 2003 IEEE International Conference on*, vol. 2, 2003, pp. II–561–4. [Online]. Available: http://dx.doi.org/10.1109/ICASSP.2003.1202428

[24] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, “Accuracy-Guaranteed Bit-Width Optimization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, October 2006.

[25] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer, “Automatic Accuracy-Guaranteed Bit-Width Optimization for Fixed and Floating-Point Systems,” in *2007 International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2007, pp. 617–620.

[26] G. F. Caffarena, “Combined Word-Length Allocation and High-Level Synthesis of Digital Signal Processing Circuits,” Ph.D. dissertation, Universidad Politécnica De Madrid, 2008.

[27] J. L. D. Comba and J. Stolfi, “Affine arithmetic and its applications to computer graphics,” in *Anais do VI Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens (SIBGRAPI’93)*, Oct. 1993, pp. 9–18.

[28] C. F. Fang, R. A. Rutenbar, M. Püschel, and T. Chen, “Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling,” in *DAC ’03: Proceedings of the 40th annual Design Automation Conference*. New York, NY, USA: ACM, 2003, pp. 496–501.

[29] G. Cybenko, “Error Analyses of Some Signal Processing Algorithms,” Ph.D. dissertation, Princeton University, Princeton, NJ, 1978.

[30] ——, “The Numerical Stability of the Levinson-Durbin Algorithm for Toeplitz Systems of Equations,” *SIAM Journal on Scientific and Statistical Computing*, vol. 1, no. 3, pp. 303–319, 1980. [Online]. Available: http://dx.doi.org/http://dx.doi.org/10.1137/0901021

[31] K. Konstantinides, V. C. Tyree, and K. Yao, “Single chip implementation of the Levinson algorithm,” *IEEE Journal of Solid-State Circuits*, vol. 20, no. 5, pp. 1072–1079, Oct. 1985.

[32] J. H. Chen, M. J. Melchner, R. V. Cox, and D. O. Bowker, “Real-time implementation and performance of a 16 kb/s low-delay CELP speech coder,” in *1990 International Conference on Acoustics, Speech, and Signal Processing, ICASSP-90*. IEEE, Apr. 1990, pp. 181–184 vol.1.

[33] A. Sanyal, S. Das, P. Venkateswaran, S. K. Sanyal, and R. N. Nandi, “An efficient time domain speech compression technique and hardware implementation on TMS320C5416 digital signal processor,” in *International Conference on Signal Processing, Communications and Networking, ICSCN ’07*. IEEE, Feb. 2007, pp. 26–29.

[34] M. Kim, J. Lee, and Y. Kim, “Implementation of the Levinson algorithm for MMSE equalizer,” in *International SoC Design Conference, 2008. ISOCC ’08.*, vol. 03. IEEE, Nov. 2008, pp. III–15–III–16.

[35] A. Sergiyenko, O. Maslennikow, P. Ratuszniak, N. Maslennikowa, and A. Tomas, *Application Specific Processors for the Autoregressive Signal Analysis*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2010, vol. 6067. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14390-8_9

[36] T. V. Huynh and M. Mücke, “Exploiting Reconfigurable Hardware to Provide Native Support of Double Precision Arithmetic on Embedded CPUs,” in *Research Poster Session, International Supercomputing Conference (ISC)*, Hamburg, Germany, 2010.

[37] T. V. Huynh, M. Mucke, and W. Gansterer, “Native double precision LINPACK implementation on a hybrid reconfigurable CPU,” in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, Anchorage, Alaska, USA, May 2011, pp. 298 –301.

[38] T. V. Huynh and M. Mücke, “Error analysis and precision estimation for floating-point dot-products using affine arithmetic,” in *2011 IEEE International Conference on Advanced Technology for Communications (ATC2011)*, Danang, Vietnam, Aug 2011, pp. 319 –322.

[39] T. V. Huynh, M. Mücke, and W. N. Gansterer, “Evaluation of the Stretch S6 Hybrid Reconfigurable Embedded CPU Architecture for Power-Efficient Scientific Computing,” in *2012 International Conference on Computational Science (ICCS)*, Omaha, Nebraska, USA, June 2012, in press.

[40] T. V. Huynh and M. Mücke, “A Tool for Floating-Point Rounding Error Modeling and Estimation in Scientific Computing Applications using Affine Arithmetic,” in *under preparation*, 2012.

[41] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, “MPFR: A multiple-precision binary floating-point library with correct rounding,” *ACM Trans. Math. Softw.*, vol. 33, no. 2, pp. 13+, June 2007. [Online]. Available: http://dx.doi.org/10.1145/1236463.1236468

[42] F. Fang, T. Chen, and R. A. Rutenbar, “Floating-point bit-width optimization for low-power signal processing applications,” in *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2002 (ICASSP ’02)*, Aug. 2002, pp. III–3208–III–3211. [Online]. Available: http://dx.doi.org/10.1109/ICASSP.2002.1005370

[43] S. Oberman, G. Favor, and F. Weber, “AMD 3DNow! Technology: Architecture and Implementations,” *IEEE Micro*, vol. 19, no. 2, pp. 37–48, March 1999. [Online]. Available: http://dx.doi.org/10.1109/40.755466

[44] *Intel SSE4 Programming Reference.*

[45] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scale, “AltiVec extension to PowerPC accelerates media processing,” *IEEE Micro*, vol. 20, no. 2, pp. 85–95, March 2000. [Online]. Available: http://dx.doi.org/10.1109/40.848475

[46] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford, UK: Oxford University Press, 2000. [Online]. Available: http://portal.acm.org/citation.cfm?id=318930

[47] W. N. Gansterer, M. Mücke, and K. Prikopa, “Arbitrary precision iterative refinement,” 2011.

[48] Stretch Inc., *Stretch SCP Programmer’s Reference - Version 1.0.* Stretch INC, 2007.

[49] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic.* Birkhäuser Boston, 2010.

[50] J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK benchmark: Past, present and future,” *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003. [Online]. Available: http://dx.doi.org/10.1002/cpe.728

[51] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 4th Edition*, 4th ed. Morgan Kaufmann, Sept. 2006. [Online]. Available: http://www.worldcat.org/isbn/0123704901

[52] R. E. Moore, *Interval Analysis.* Prentice-Hall, Englewood Cliffs N. J., 1966.

[53] J. Stolfi and L. H. Figueiredo, “Self-validated numerical methods and appications,” in *Brazilian Mathematics Colloquium Monograph*, Rio de Janeiro, Brazil, 1997.

[54] D. P. Bertsekas and J. N. Tsitsiklis, *Introduction To Probability.* Athena Scientific, 2002.

[55] Y. Voronenko and M. Puschel, “Mechanical Derivation of Fused Multiply-Add Algorithms for Linear Transforms,” *IEEE Transactions on Signal Processing*, vol. 55, no. 9, pp. 4458–4473, Sept. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSP.2007.896116

[56] T. Ogita, S. M. Rump, and S. Oishi, “Accurate sum and dot product,” *SIAM J. Sci. Comput.*, vol. 26, no. 6, pp. 1955–1988, June 2005. [Online]. Available: http://dx.doi.org/10.1137/030601818

[57] S. Boldo and J. M. Muller, “Some Functions Computable with a Fused-MAC,” in *17th IEEE Symposium on Computer Arithmetic (ARITH’05).* Washington, DC, USA: IEEE, 2005, pp. 52–58. [Online]. Available: http://dx.doi.org/10.1109/ARITH.2005.39

[58] S. Boldo and J.-M. Muller, “Exact and Approximated Error of the FMA,” *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 157–164, February 2011. [Online]. Available: http://dx.doi.org/10.1109/TC.2010.139

[59] S. Graillat, P. Langlois, and N. Louvet, “Algorithms for accurate, validated and fast polynomial evaluation,” *Japan Journal of Industrial and Applied Mathematics*, vol. 26, no. 2, pp. 191–214, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1007/BF03186531

[60] P. Langlois and N. Louvet, “Operator Dependant Compensated Algorithms,” in *12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN).* IEEE, 2006, pp. 2–2. [Online]. Available: http://dx.doi.org/10.1109/SCAN.2006.36

[61] R. K. Montoye, E. Hokenek, and S. L. Runyon, “Design of the IBM RISC system/6000 floating-point execution unit,” *IBM J. Res. Dev.*, vol. 34, no. 1, pp. 59–70, 1990. [Online]. Available: http://dx.doi.org/10.1147/rd.341.0059

[62] E. Quinnell, E. E. Swartzlander, and C. Lemonds, “Bridge Floating-Point fused Multiply-Add design,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 12, pp. 1727–1731, Dec. 2008. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2008.2001944

[63] J. D. Pryce and G. F. Corliss, “Interval arithmetic with containment sets,” *Computing*, vol. 78, no. 3, pp. 251–276, Nov. 2006. [Online]. Available: http://dx.doi.org/10.1007/s00607-006-0180-4

[64] G. H. Golub and C. F. Van Loan, *Matrix computations (3rd ed.).* Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[65] T. F. Chan and P. C. Hansen, “A look-ahead Levinson algorithm for general Toeplitz systems,” *IEEE Transactions on Signal Processing*, vol. 40, no. 5, pp. 1079–1090, May 1992.

[66] P. Delsarte and Y. Genin, “The Split Levinson Algorithm,” *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 34, no. 3, pp. 470–478, June 1986.

[67] J. Le Roux and C. Gueguen, "A fixed point computation of partial correlation coefficients in linear prediction," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 2. IEEE, May 1977, pp. 742–743.

[68] C. P. Rialan and L. L. Scharf, "Fixed-point error analysis of the lattice and the Schur algorithms for the autocorrelation method of linear prediction," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 37, no. 12, pp. 1950–1957, Dec. 1989.

[69] P. Delsarte and Y. Genin, "On the splitting of classical algorithms in linear prediction theory," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 35, no. 5, pp. 645–653, May 1987.

[70] J. R. Bunch, "Stability of Methods for Solving Toeplitz Systems of Equations," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 2, pp. 349–364, 1985.

[71] S. Chandrasekaran, M. Gu, X. Sun, J. Xia, and J. Zhu, "A Superfast Algorithm for Toeplitz Systems of Linear Equations," *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 4, pp. 1247–1266, 2008. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/040617200

[72] R. Viswanathan and J. Makhoul, "Quantization properties of transmission parameters in linear predictive systems," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 23, no. 3, pp. 309–321, June 1975. [Online]. Available: http://dx.doi.org/10.1109/TASSP.1975.1162675

[73] C. N. Papaodysseus, E. B. Koukoutsis, and C. N. Triantafyllou, "Error sources and error propagation in the Levinson-Durbin algorithm," *IEEE Transactions on Signal Processing*, vol. 41, no. 4, pp. 1635–1651, 1993.

[74] *http://www.ldc.upenn.edu/*, (last accessed: July 06, 2012).

[75] K. K. Paliwal and B. S. Atal, "Efficient vector quantization of LPC parameters at 24 bits/frame," *IEEE Transactions on Speech and Audio Processing*, vol. 1, no. 1, pp. 3–14, Jan. 1993.

[76] F. de Dinechin and B. Pasca, "Custom arithmetic datapath design for FPGAs using the FloPoCo core generator," *Design & Test of Computers, IEEE*, vol. 28, no. 4, pp. 18– 27, Jul.-Aug. 2011. [Online]. Available: http://dx.doi.org/10.1109/MDT.2011.44

[77] *FloPoCo*, (last accessed: July 06, 2012). [Online]. Available: http://flopoco.gforge.inria.fr/

[78] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[79] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Formal datapath representation and manipulation for implementing DSP transforms," in *Design Automation Conference (DAC)*, 2008, pp. 385–390.

[80] *SPIRAL*, (last accessed: July 06, 2012). [Online]. Available: http://www.spiral.net/

— End —