THE UNIVERSITY OF SYDNEY

COPYRIGHT AND USE OF THIS THESIS

THE UNIVERSITY OF
SYDNEY

SCHOOL OF ELECTRICAL & INFORMATION ENGINEERING

COMPUTER ENGINEERING LABORATORY

Automated Dynamic Error Analysis Methods for Optimization of Computer Arithmetic Systems

Michael FRECHTLING B.E. (Hons)

A thesis submitted in fulfilment of

the requirements for the degree of

Doctor of Philosophy

15th September 2015

For my parents

Abstract

Computer arithmetic is one of the more important topics within computer science and engineering. The earliest implementations of computer systems were designed to perform arithmetic operations and most if not all digital systems will be required to perform some sort of arithmetic as part of their normal operations. This reliance on the arithmetic operations of computers means the accurate representation of real numbers within digital systems is vital, and an understanding of how these systems are implemented and their possible drawbacks is essential in order to design and implement modern high performance systems. At present the most widely implemented system for computer arithmetic is the IEEE754 Floating Point system, while this system is deemed to the be the best available implementation it has several features that can result in serious errors of computation if not implemented correctly. Lack of understanding of these errors and their effects has led to real world disasters in the past on several occasions. Systems for the detection of these errors are highly important and fast, efficient and easy to use implementations of these detection systems is a high priority. Detection of floating point rounding errors normally requires run-time analysis in order to be effective. Several systems have been proposed for the analysis of floating point arithmetic including Interval Arithmetic, Affine Arithmetic and Monte Carlo Arithmetic. While these systems have been well studied using theoretical and software based approaches, implementation of systems that can be applied to real world situations has been limited due to issues with implementation, performance and scalability. The majority of implementations have been software based and have not taken advantage of the performance gains

associated with hardware accelerated computer arithmetic systems. This is especially problematic when it is considered that systems requiring high accuracy will often require high performance. The aim of this thesis and associated research is to increase understanding of error and error analysis methods through the development of easy to use and easy to understand implementations of these techniques.

Acknowledgements

I would like to express my sincere thanks and appreciation to my supervisor and mentor, Professor Philip H.W. Leong. Your advice, encouragement and direction on my research, my thesis and my career have been invaluable over the last four years and I am extremely grateful to have had the opportunity to work with you. I would also like to thank my associate supervisor, Professor Craig Jin, whose advice and assistance has also been invaluable. I would especially like to thank the past and present members of the Computer Engineering Laboratory and CARLab, in particular Mr. Nicholas Fraser, Ms. Calla Klafas, Mr. Duncan Moss, Mr. Stephen Tridgell, Dr. Nicolas Epain, Dr. Aengus Martin, Dr. Abhaya Parthy and Dr. Andrew Wabnitz. Your ability to provide assistance, advice and, most importantly, distractions have helped in making the last four years survivable.

A special thanks to my family. Words cannot express how grateful I am to my parents, brothers and extended family for all of the sacrifices that you've made on my behalf and whose love and support have made this work possible. I would also like to thank all of my friends who supported me in my work, and encouraged me to strive towards my goal.

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work and contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at the University of Sydney or any other educational institution, except where due acknowledgment is made in the thesis.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged, specifically:

- The research direction and subject were suggested by Professor Philip H.W. Leong.
- Advice and editing assistance for the preparation of this thesis has been provided by Professor Philip H.W. Leong.
- The **cilly** source-to-source compiler as presented in Chapter 4 was developed by Professor Philip H.W. Leong.
- Monte Carlo arithmetic was first proposed by Professor D.S. Parker at UCLA [142].

Contents

List of Figures

List of Tables

List of Acronyms

AA	affine arithmetic
ALU	arithmetic logic unit
ASA	adaptive simulated annealing
ATMCA	Monte Carlo arithmetic with antithetic variates
AXI	advanced extensible interface
ASIC	application specific integrated circuit
BCD	binary coded decimal
BFGS	Broyden, Fletcher, Goldfarb, Shanno
BLAS	basic linear algebra subprograms
BRAM	block RAM
BSP	board support package
CADNA	control of accuracy and debugging for numerical applications
CESTAC	Contrôle et Estemation Stochastique des Aarondis Calculs
CFG	control flow graph
CIL	C intermediate language
CPU	central processing unit

DFG data flow graph

DMA direct memory access

DSA discrete stochastic arithmetic

DSP digital signal processing/processor

EDSAC electronic delay storage automatic calculator

EDVAC electronic discrete variable automatic computer

ENIAC electronic numerical integrator and computer

FDIV floating point division

FF flip flop

FLOPS floating point operations per second

FMA fused multiply add

FMAC fused multiply accumulate

FP floating point

FPGA field programmable gate array

FPL field programmable logic

FPU floating point unit

FSM finite state machine

GCC GNU compiler collection

GPROF GNU profiler

GPU graphics processing unit

HW hardware

HDL hardware description language

HLS high level synthesis

HPC high performance computing

IA interval arithmetic

IIA inner interval arithmetic

IEEE Institute of Electrical and Electronic Engineers

IID independent and identically distributed

IO input/output

IRLS iteratively re-weighted least squares

L-BFGS limited memory BFGS

LAPACK linear algebra package

LDS low discrepancy sequence

LINPACK linear equations software package

LSB least significant bit

LUT look up table

LZD leading zero detector

MAC multiply accumulate

MCA Monte Carlo arithmetic

MCALIB Monte Carlo arithmetic library

MCM Monte Carlo method

MCMCM Markov chain Monte Carlo method

MCP Monte Carlo programming

MECTG maximally equi-distributed combined Tausworthe generator

MILP mixed integer linear programming

MPFR Multiple Precision Floating-Point Reliably

MSB most significant bit

MSE mean squared error

MWLP multiple word length paradigm

NaN not a number

OS operating system

PCIe peripheral component interconnect express

PDF probability density function

PL programmable logic

PRNG psuedo-random number generator

PS processing system

qMCM quasi-Monte Carlo method

qMCA quasi-Monte Carlo arithmetic

qNaN quiet not a number

RAM random access memory

RIA random interval arithmetic

RqMCM randomized quasi-Monte Carlo method

RSD relative standard deviation

RTL register transfer level

SDLC software development life cycle

SDK software development kit

SIMD single instruction multiple data

sNaN signaling not a number

SNR signal to noise ratio

SoC system on a chip

SW software

ULP unit in the last place

VHDL very high speed integrated circuit hardware description language

VHSIC very high speed integrated circuit

VLSI very large scale integration

XSC extensions for scientific computation

Chapter 1

Introduction

1.1 Motivation & Aims

Computer arithmetic is an umbrella term for the systems and techniques used to perform arithmetic operations in digital systems. Computer arithmetic systems have evolved into sophisticated real number formats such as floating point (FP) arithmetic. During this evolution they have become an integral part of modern society and the number of commercial, industrial and scientific applications requiring computer arithmetic has grown significantly. Due to the requirement to perform large numbers of calculations at high speed these arithmetic systems have evolved to the point where performance of billions of operations a second is easily obtainable by the average user, and modern supercomputers have achieved petascale performance [65, 141]. The earliest innovations in computer arithmetic involved the development of basic binary arithmetic operations, these simple implementations of boolean algebra allowed for the development of integer arithmetic systems that allowed digital systems to perform simple mathematical operations such as add and subtract on whole numbers. A major breakthrough in the development of computer arithmetic was the invention of methods for handling real numbers, the most widely implemented of which is FP arithmetic. FP arithmetic uses sign-magnitude representation, where a number is represented using a sign bit s, an exponent field e, and a

significand m, (also known as a mantissa). Using this representation the true value is $(-1)^s.m.\beta^e$ where β is the radix of the storage number system, (the most common of which is binary with a radix $\beta = 2$).

In developing real number systems a fundamental issue is raised in representing numbers that require high levels of precision and the accuracy with which these numbers can be processed. The precision of a number system is defined as the number of base-β digits used to represent the fractional part of a number. Typical notations will use the term p to represent the precision of a number system, in fixed point systems p represents the number of digits used to store the fraction part of the number, and in FP systems p represents the number of digits used to store the significand. By bounding the precision of these number system to a finite quantity, the number of unique representations achievable by that number system is also bounded. In a number system using a total of n digits for storage, a total of β^n unique representations are possible. When this metric is applied to the infinite set of real numbers, it is obvious that some numbers are representable, and some are not. This leads to the concept of **Exact** versus **Inexact** values, where exact values are those that can be represented exactly by a number system, while inexact values cannot be represented and must be rounded to an appropriate exact value. The difference between exact and inexact values and the subsequent requirement for rounding leads to rounding error within computer arithmetic operations. This can compound and transform results of operations in ways that can lead to significant error in the results.

The field of error analysis has grown out of a desire to measure the effects of rounding error on computer arithmetic calculations and to develop better methods for performing these operations. Much research has been conducted in the field for a number of decades, with seminal work being published as early as 1959 by computer scientists and numerical analysts such as Carr [18], Wilkinson [170, 171], Goldberg [64] and Kahan [93]. This has lead to the development of practical analytic

techniques for measuring rounding error in individual applications and software and allowed developers to take steps to predict and avoid real world problems that can result from arithmetic error. These techniques are used in a variety of scientific, commercial and industrial applications such as financial engineering [63], analogue circuit design [44], development of safety critical software as used in avionics [38, 39] and medicine [147]. Techniques for error analysis are broadly split into static program analysis, techniques that analyse an abstract model of an algorithm without the need for execution, and dynamic program analysis, which performs analysis on the results of an execution.

While the field of error analysis is well studied the application of error analysis has been limited thus far, with applications generally being found in high level industrial and scientific fields and in safety critical software. One of the primary reasons for this limited implementation is that error analysis techniques tend to be highly complex and understanding of error analysis theory tends to be limited to specialists in the fields of computer architecture, computer arithmetic and numerical analysis. Dynamic error analysis techniques can require significant changes to existing source code and due to the requirement for extended precision or changes to memory structures these systems are often implemented in software rather than hardware impacting performance. Static analysis techniques are also difficult to implement due to their requirement for in depth understanding of the mathematical models involved, and do not scale well beyond small sub-routines [96]. In order to address these issues we have endeavoured to develop automated methods that simplify the implementation of dynamic error analysis methods and aim to achieve the following:

- Develop methods to allow developers to understand rounding error and required precision level.

- Automate the application of Monte Carlo arithmetic (MCA) analysis.

- Reduce the computational requirements of error analysis techniques.

1.2 Contributions

In order to achieve these goals methods for automating the application of MCA have been investigated and the contributions of the work include the following:

- A novel MCA co-processor architecture using standard FP cores [58].
- The first complete hardware (HW) accelerated implementation of MCA for run-time error analysis [58].
- An open source MCA implementation capable of performing variable precision MCA and supporting both single and double precision FP formats [57].
- An investigation into the application of variance reduction techniques to MCA in order to improve performance.

1.3 Organization of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 provides a review of existing work on computer arithmetic and error analysis, and review of Monte Carlo method (MCM)s and MCA is presented in Chapter 3. Work conducted on the Monte Carlo arithmetic library (MCALIB) is presented in Chapter 4 and work conducted on a field programmable gate array (FPGA) based MCA floating point unit (FPU) including the system architecture, testing procedure and results is presented in Chapter 5. Chapter 6 presents work on the application of variance reduction methods to MCA, and finally conclusions are drawn in Chapter 7.

Chapter 2

Background

2.1 Introduction

Computer arithmetic is one of the more important, and yet under-appreciated, topics within computer science and engineering. Some of the earliest implementations of computer systems, such as the electronic numerical integrator and computer (ENIAC), electronic discrete variable automatic computer (EDVAC) and electronic delay storage automatic calculator (EDSAC), were designed solely to perform mathematical calculations [150], and the ability to perform accurate and efficient mathematical operations is a fundamental requirement of any modern system. However, computer arithmetic systems are often assumed to be a perfectly accurate and fail-safe component by both users and developers. Implementations of computer arithmetic systems have progressed from simple adders and subtracters capable of performing several operations a minute [149], to modern multi-core processors capable of performing hundreds of billions of floating point operations per second (FLOPS) and super-computers capable of even higher performance.

2.2 Computer Arithmetic

2.2.1 Binary Number System & Integer Arithmetic

The binary number system is a base-2 system that represents numeric values using two values, 0 and 1, or more formally:

The binary (base-2) number system is a positional notation system with a radix $\mathbb{B} = 2$

The binary number system is of particular use in the field of computing due to the use of logic gates to implement digital logic and circuity, and the binary number system has become the standard for the representation of digital logic and computer arithmetic. The underlying theory of boolean logic and the original use of binary for its representation was laid out by *Claude E. Shannon* in 1938 in his masters thesis and the associated journal article [157], this paper has since become one of the foundations of practical digital circuit design.

In terms of computer arithmetic, the binary number system is used to represent decimal values in a way that can be stored in a computer system, an encoding system referred to as binary coded decimal (BCD). Using this representation decimal values are converted to binary and vice-versa, and basic operators such as addition and subtraction can be implemented using digital logic. An n-bit binary value is converted to decimal as follows [118]:

$$x = \sum_{i=0}^{n-1} b_i 2^i \tag{2.1}$$

where b_i represents a non-zero bit in place i. This representation is the most basic form of binary numerical representation and is used to implement *integer arithmetic*, as the system is only able to represent integer values in this form. Using this format the range of values representable by an n-bit number is $[0, 2^n - 1]$.

Input Values		Output Values		Decimal Result
a	b	c_o	r_o	r
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	2

Table 2.1: Integer Arithmetic - Half adder truth table

2.2.2 Basic Operations

Basic operators for use in binary integer arithmetic are built from fundamental boolean operations, and as such the an operation on an n-bit binary value is built by connecting together logic designed to perform the operation on 1-bit values. The most basic binary operation is binary addition, and when performed on 1-bit values must produce the following results:

$$0 + 0 = 0 \tag{2.2}$$

$$0 + 1 = 1 \tag{2.3}$$

$$1 + 0 = 1 \tag{2.4}$$

$$1 + 1 = 10 \tag{2.5}$$

The final operation listed above introduces the idea of a *carry* bit. In this case the result of a 1-bit operation has produced a 2-bit result, i.e. the operation has overflowed and the precision level of the operation, (one in this case), is no longer sufficient to represent the result [118]. In this case the result of the operation $1 + 1$ will be 0, with a *carry out* value of 1. A 1-bit binary adder unit with a carry out signal is referred to as a *Half Adder* [71] and is implemented according to the truth table shown in table 2.1, and the following boolean arithmetic:

$$r_o = a \oplus b \tag{2.6}$$

$$c_o = a \bullet b \tag{2.7}$$

Input Values			Output Values		Decimal Result
a	b	c_i	c_o	r_o	r
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	2
1	0	0	0	1	1
1	0	1	1	0	2
1	1	0	1	0	2
1	1	1	1	1	3

Table 2.2: Integer Arithmetic - Full adder truth table

where a and b are 1-bit input signals, r_o is the result bit, c_o is the carry bit and $\bullet$, $\oplus$ represent the boolean operations AND and XOR respectively. The implementation of a carry out signal leads to the corresponding implementation of a *carry in* signal, allowing for an addition operation to be implemented that accounts for a carry signal that has propagated through a chain of adder modules. A 1-bit adder unit that implements both carry in and carry out signals is called a *Full Adder* [71] and is implemented according to the truth table in table 2.2, and the following boolean logic:

$$r_o = (a \oplus b) \oplus c_i \tag{2.8}$$

$$c_o = (a \bullet b) + (c_i \bullet (a \oplus b)) \tag{2.9}$$

where a and b are 1-bit input signals, r_o is the result signal, c_o and c_i are the carry out and carry in signals, and $+$ represents the boolean OR operation. An n-bit binary adder is implemented by connecting n full adders together, linking the carry out signal of adder n to the carry in signal of adder $n+1$, with the carry in signal of the first adder being locked to 0 and the carry out signal of the final adder used to indicate overflow. This set-up is referred to as a *Ripple Carry Adder* [11] and is a fundamental building block in most arithmetic systems. Although the ripple carry adder is simple to implement the design leads to a high critical path delay as each full adder in the chain must wait for the carry bit to be calculated by the previous

adder. The delay leads to poor performance and as such alternate methods for integer arithmetic are often used. In the case of carry look-ahead adders [11] the carry bit is calculated before the sum bit reducing the wait time in larger adder chains. In the case of carry-select and carry-bypass adders multiple additions are performed with each addition making different assumptions about the behaviour of the carry bits, the results are then multiplexed based on the actual behaviour of the carry bits to produce the correct result [10, 72]. While these methods provide improved performance over the standard ripple-carry method, they are more costly in terms of area use. Binary subtraction is implemented using the same methods as listed is this section, but negating one of the operands by finding the radix complement of the operand. The theory and methods of finding an operands complement are discussed in the next section.

Binary multiplication is a more complex operation to implement, especially in a way that provides high performance. The basic algorithm for performing binary multiplication involves adding together a set of partial products in a similar fashion to decimal multiplication. To multiply two n-digit numbers a and b together a total of n partial products are calculated by shifting the multiplicand a to the left i places and multiplying by the i^{th} digit of the multiplier b, the n partial products are then summed to find the final product;

$$a \cdot b = \sum_{i=0}^{n} (a \ll i) \cdot (b_i) \tag{2.10}$$

where $\ll$ represents the **SLL!** (**SLL!**) operation. This method is shown below where the product of 5 (binary value 101) and 7 (binary value 111) is calculated:

$$111 \cdot 101 = ((111 \ll 2) \cdot 1) + ((111 \ll 1) \cdot 0) + ((111 \ll 0) \cdot 1) \tag{2.11}$$

$$= 011100 + 000000 + 000111 \tag{2.12}$$

$$= 100011 \tag{2.13}$$

The result of multiplying two n-bit numbers will produce a result with up to $2n$-bits, and implementations often require a set of two registers, (high and low result registers), to store the result of the operation. The majority of computer arithmetic systems did not implement multiplication instructions until the late 1970s [35], relying instead on shift-accumulate routines performed in software. Early implementations of multiplier circuits used a set of shifters coupled with an accumulator to sum one partial product per cycle. Modern implementations of multipliers use methods such as the Baugh-Wooley algorithm [9], Wallace Trees [167] or Dadda multipliers [32] to perform the required addition operations within one cycle.

The final fundamental operation is binary division, where given a dividend and a divisor the operation finds the whole number quotient and the remainder:

$$N = Q \cdot D + R \tag{2.14}$$

where N is the dividend (numerator), Q is the quotient, D is the divisor (denominator) and R is the remainder. As whole number division produces two whole number results, two separate instructions are typically implemented, the divide instruction, (/), returns the quotient, while the modulo instruction, (%), returns the remainder:

$$Q = N/D \tag{2.15}$$

$$R = N\%D \tag{2.16}$$

Algorithms for division are divided into two categories, slow algorithms which perform an n-digit division in n steps, (where n is the number of digits in the quotient), and fast division algorithms, which reduce the number of steps required to find the results. The most common slow division methods are the restoring and non-restoring division methods, both of which are based on long division methods

and implement the following standard recurrence equation:

$$P_{j+1} = (R \cdot P_j) - (Q_{n-(j+1)} \cdot D) \tag{2.17}$$

where P_j is the partial remainder, and $Q_{n-(j+1)}$ is the digit of the quotient in position $n-(j+1)$ indexed from least significant bit (LSB) to most significant bit (MSB). Fast algorithms do not typically implement integer division but rather real number division and as such are discussed in later sections on fixed and floating point operators.

2.2.3 Signed Representation and Complement Operations

A significant characteristic of any number system is the way in which that system represents negative values. A basic binary representation using an n-bit value will only be able to represent positive integers, and as detailed in previous sections can represent 2^n unique values with a range of $[0, 2^n - 1]$. To perform real world calculations a number system must represent both positive and negative values, and because standard numeric sign operators, ($+$ and $-$ symbols), are not available sign indicators must be implemented using binary digits [80]. Two basic implementations of signed representations have been developed, sign magnitude representation and the method of complements. Signed magnitude representation requires a single bit of the value (typically the MSB) to be dedicated as the *sign bit* of the value, with negative values indicated by a sign value of 1 and positive values indicated by a sign value of 0. For an n-bit signed value there will be 1 sign bit followed by $n-1$ value bits. This results in 2^n unique values with a range of $[-2^{n-1} - 1, 2^{n-1} - 1]$ including both positive and negative zero values [131].

The alternative to signed representation is the method of complements, a method for signed representation that also allows subtraction operations to be performed using addition operators [103]. The method of complements is broken down into

two representations, the radix complement and the diminished radix complement. The radix complement $\hat{x}$ of a base-β number x is defined as follows:

$$\hat{x} = \beta^n - x \tag{2.18}$$

and the diminished radix complement is defined as:

$$\hat{x} = (\beta^n - 1) - x \tag{2.19}$$

For the purposes of binary integer arithmetic the radix complement of a binary number is the two's complement, and the diminished radix complement is the one's complement. Finding the two's complement of a binary number is simplified by first determining the one's complement then adding 1 to the result. As the value $\beta^n - 1$ corresponds to the value β^n repeated n times, as such finding the diminished radix complement is done by subtracting the value β^n from each digit, in the case of one's complement this can be done by switching the values of each digit from 0 to 1 and vice-versa, adding 1 to the result produces the two's complement form, for example negating the value 0101, (decimal 5), as shown below:

$$x = 0101 \tag{2.20}$$

$$\text{complement}_1(x) = 1010 \tag{2.21}$$

$$\text{complement}_2(x) = 1011 \tag{2.22}$$

where complement_1 and complement_2 represent the one's and two's complement operations respectively. Using this system a positive value will be indicated by a zero in the MSB, and negatives a one in the MSB [102]. Using the method of complements binary subtraction can be performed by finding the negative value of a number and performing a binary addition to calculate the results. Using the previous example of

0101, (decimal 5), we can calculate $10 - 5$ as follows:

$$x = 10 - 5 \tag{2.23}$$
$$= 10 + (-5) \tag{2.24}$$
$$= 1010 + 1011 \tag{2.25}$$
$$= 10101 \tag{2.26}$$

The result of the addition operation corresponds to the result of calculating $x + (\beta^n - y)$, in order to find the correct result the β^n term must be removed corresponding to $x + (\beta^n - y) - \beta^n = x - y$. If it is assumed that $x \geq y$, the the result of the subtraction is always greater than or equal to β^n, and truncating the leading one of the result is equivalent to subtracting β^n. Removing the leading one from the last stage of the previous example produces the correct result 0101.

At present arithmetic systems will use either two's complement or sign magnitude representation to represent negative values. Two's complement is preferred for integer arithmetic due to its simplicity, (no changes are required to standard arithmetic units), and the systems lack of a value for -0. While sign magnitude is preferred for use in more complex arithmetic systems such as floating point (FP) arithmetic.

2.3 Fixed Point Arithmetic

Fixed point arithmetic is a number system for representing real numbers ($x \in \mathbb{R}$) containing both a whole and fractional part, as opposed to binary integers. Fixed point is one of the earliest attempts to find a suitable implementation of real number arithmetic for computer systems. During the design of early computer systems some of the earliest implementations of computer arithmetic were developed. During this time designers considered the benefits of different types of arithmetic systems, considering binary and decimal arithmetic systems, and fixed v. floating point

systems. As computer systems have further developed FP has become the default arithmetic system and fixed point arithmetic is limited to systems were floating point unit (FPU)s are not available, in applications were minimizing computational complexity is paramount or in digital signal processing/processor (DSP) systems were the range is fixed.

2.3.1 Format and Basic Notation

Fixed point numbers are denoted as such as the whole and fractional parts of the format are fixed for a particular format or implementation. For a given implementation of an n-bit fixed point number system the whole and fractional parts of the number are designated as a integer bits and b fractional bits such that $a = n - b$. Like FP numbers, fixed point numbers are stored as n-bit binary words and conversion to decimal representation can be performed using the formats scaling factor, given as $\frac{1}{2^b}$ for binary systems with radix $\beta = 2$;

$$x = \hat{x} \cdot 2^{-b} \tag{2.27}$$

The decision of what value to use for the scaling factor is an extremely important one to make during the design of a fixed point number system as this value determines the range of values that can be represented by the system. The range of an unsigned fixed point number $\mathbb{U}(a, b)$ is given as;

$$0 \leq x \leq 2^a - 2^{-b} \tag{2.28}$$

Using signed representation an n-bit signed binary Fixed Point value $\mathbb{U}(s, a, b)$ represents a real value with a sign $s \in \{0, 1\}$, using a integer bits and b fractional bits where $a, b \in \mathbb{Z}, b \neq 0$ and $a + b = n - 1$. The value can be converted to decimal

form using the following [180]:

$$x = (-1)^s \left[\sum_{i=0}^{a+b-1} u_i \right] 2^{-b} \tag{2.29}$$

and the range of a signed fixed point system can be calculated as follows:

$$-2^a \leq x \leq 2^a - 2^{-b} \tag{2.30}$$

2.3.2 Basic Operators

Basic operations for fixed point arithmetic (addition, subtraction, multiplication and division) are performed using standard binary arithmetic (e.g. ripple carry adders for addition) and do not required extra logic in order to perform the operations, however, several rules must be considered in order to ensure that the operands and the operation are compatible and the result is valid.

1. **Addition and Subtraction:** In order to perform addition/subtraction of two fixed point values those values must be scaled for the result to be valid, i.e both numbers must use the same scaling factor b and format (signed v. unsigned). This concept is defined more formally by stating that an operation on two fixed point values $X(s,a,b) \pm Y(s,c,d)$ will produce a valid result if $a = c$ and $b = d$. The resulting value can overflow (carry) requiring a format $R(s,a+1,b)$ to handle all possible results, the range of possible results is calculated by:

$$x = X(s,a,b) \pm Y(s,a,b) \tag{2.31}$$

$$x \in \left[-2^{a+1}, 2^{a+1} - 2^{-b} \right] \tag{2.32}$$

The increase in precision required due to a possible carry will result in $n+1$ bits being required to add two n bit values, as in standard binary arithmetic. Issues from overflow and loss of precision are covered in more detail in section 2.3.3. In the event that the two operands have different scaling factors one value must

be scaled, (aligning of the decimal point), in order to perform the operation correctly. Given two fixed point values $x \in U(s,a,b)$ and $y \in U(s,c,d)$ the value x may be scaled by multiplying by a scaling factor $\frac{2^b}{2^d}$, and alternatively y may be scaled by multiplying by $\frac{2^d}{2^b}$. In practise this multiplication my be performed using a shift operation to shift the value $x \ll b-d$ or shifting the value $y \ll d-b$.

2. **Multiplication:** Fixed point multiplication operations are performed using standard binary multiplication techniques. For an operation $X(s,a,b) \cdot Y(s,c,d)$ that multiplies two n-bit numbers the result will contain up to $2n$ bits, with the possibility of overflow occurring requiring $2n+1$ bits to represent all possible results. The range of possible results is calculated as follows:

$$x = X(s,a,b) \cdot Y(s,c,d) \tag{2.33}$$

$$x \in \left[-2^{a+c+1}, 2^{a+c+1} - 2^{-(b+d)}\right] \tag{2.34}$$

In the case of multiplication the increase in precision from the operation affects both the integer part and the fraction part of the number, doubling the length of both values. This is problematic as the value must be rounded to reduce the scaling factor to the original level in order to use the result in further operations. As fixed point arithmetic is not widely implemented in modern computer systems or programming languages no standard is available with definitions for rounding methods and the methods used will vary from system to system.

3. **Division:** As in the case of multiplication signed fixed point division is performed using standard binary division techniques and the resulting value will contain twice the number of bits as the operands. The range of possible results

is calculated as follows:

$$x = \frac{X(s,a,b)}{Y(s,c,d)} \tag{2.35}$$

$$x \in \left[-2^{a+c+1}, 2^{a+c+1} - 2^{-(b+d)}\right] \tag{2.36}$$

As the result of the division operation will also contain twice the number of bits as the original operands the result must be rounded in order to conform with the original fixed point format. Fixed point division techniques can be performed using slow division techniques as discussed in section 2.2.2 or using fast division techniques. The most common technique in use utilises the Newton-Raphson method, an iterative method for finding the roots of a real-valued function [182]. The basic method for finding the solution to a non-linear function $f(x) = 0$ using the Newton-Raphson method requires finding an initial estimate for the solution x_0 then refining that solution using the following iterative process:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{2.37}$$

The iterative process is repeated until the result agrees with the previous result to a pre-determined accuracy level, $x_{i+1} \approx x_i$, or until a maximum number of iterations is reached. To apply the method to division to find the result of $\frac{y}{z} = y \cdot \frac{1}{z}$ the function $f(x) = \frac{1}{x} - z$ is used to find the reciprocal at the zero of the function $x = \frac{1}{z}$, applying the formula for Newton-Raphson the following equation is found:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{2.38}$$

$$= x_i - \frac{\frac{1}{x_i} - z}{-\frac{1}{x_i^2}} \tag{2.39}$$

$$= x_i + x_i(1 - zx_i) \tag{2.40}$$

Using this formula the result can be calculated from the initial estimate using only addition and multiplication operations. Alternative fast division methods include Goldschmidt division [66], an iterative method that performs division by multiplying the dividend and the divisor by a series of common factors F_i so that the divisor, D, converges to 1, and the dividend, N converges to the solution, Q. The iterative process is repeated until the solution is suitably accurate, or a maximum number of iterations, k, is reached;

$$Q = \frac{N}{D}\frac{F_1}{F_1}\frac{F_{...}}{F_{...}}\frac{F_k}{F_k} \tag{2.41}$$

The iterative process is performed as follows:

- Determine a value for the scaling factor $F_{i+1} = 2 - D_i$
- Multiply the system by the scaling factor to determine the value of the next iteration, $\frac{N_{i+1}}{D_{i+1}} = \frac{N_i}{D_i}\frac{F_{i+1}}{F_{i+1}}$
- Checking for convergence; either $D_{i+1} = 1$ or $i = k$

2.3.3 Overflow and Precision Loss

As stated in the previous sub-section operations performed using fixed point arithmetic will often result in an increase in the number of bits required to represent the resulting value. This will result in both overflow, in which case the result is not representable in the current format, or loss of precision, where the result can be represented but some information will be lost. In the case of signed addition a one-bit carry can occur, as the MSB is reserved for the sign bit the carry cannot be included in the result and overflow will occur. In FP arithmetic this result could be shifted to re-align the value and the exponent modified, but in fixed point this value cannot be represented and the result will be incorrect. The handling of overflow exceptions in fixed point is limited to providing overflow flags to indicate an exception has occurred and the result cannot be used. Multiplication and division operations can result in not only an increase in the number of bits required to represent the integer

part of the number, a, but also an increase in the number of bits required to represent the fractional part, b. If value of b for the result is large than value for b defined for the result, $b_r > b$, then the result is either rounded or truncated, this will result in a loss of precision measurable using the absolute error form. Given a rounded result, $\hat{x}$, of an exact value, x, the absolute error ϵ is defined as follows:

$$\epsilon = |x - \hat{x}| \tag{2.42}$$

$$\leq 2^{-b_r} \tag{2.43}$$

If the result contains a value for a that is larger than the value for a defined for the system, ($a_r > a$) then the result is not representable using the current format. Some fixed point systems handle overflow by forcing results beyond the range of the system to default to the largest representable value, (either positive or negative depending on the operation and sign of the operands), a technique referred to as *saturation*.

2.4 Floating Point Arithmetic

FP arithmetic is the most widely implemented system for the approximation of real numbers in modern computer systems [22, 53, 64, 65, 95, 96, 126, 131]. The first modern example of floating point arithmetic was implemented by Konrad Zuse for the Z1 mechanical computer [131], completed in 1938. The Z1 was designed as a FP adder and subtracter, with control logic allowing for the implementation of multiplication and division [149]. The system used a 22-bit FP representation and was capable of performing one addition operation every five seconds.

FP systems are so named as the radix point, (sometimes referred to as the *decimal* or *binary* point), can be shifted relative to the significant digits of the number. The position of the radix, the significant digits of the number and the sign are stored using sign, exponent and significand values in a similar fashion to *scientific notation*.

In its simplest form a FP number $\mathbb{F}(s, m, \beta, e)$ can be expressed in the following form:

$$x = (-1)^s m \beta^e \tag{2.44}$$

where $s \in \{0, 1\}$ is the sign, β the radix (base) of the floating point system, m is the significand (significand) such that $|m| < \beta$ and e is the exponent [131]. A more formal definition of a FP system will also require definitions for the following values:

1. Precision p, the number of significant digits in the significand.
2. Maximum and minimum exponent values, e_{min} and e_{max}.
3. The exponent offset e_o

The values of e_{max} and e_{min} denote the maximum and minimum possible values of the systems exponent and are used to differentiate between zero, normalized, sub-normal and infinite numbers. The exponent offset is used to adjust the unsigned exponent value to allow for negative exponent values such that;

$$x = (-1)^s m \beta^{e - e_o} \tag{2.45}$$

$$= (-1)^s m \beta^{e_b} \tag{2.46}$$

where e is the stored exponent value and $e_b = e - e_o$ is the biased exponent value. The value of the exponent offset depends on the number of bits reserved for the exponent, w;

$$e_o = \sum_{i=0}^{w-2} \beta^i \tag{2.47}$$

The precision of a number system is an important concept in computer arithmetic as the storage capabilities of real-world computer systems limit the representation of numerical values to a finite subset of real numbers $\mathbb{F} \subset \mathbb{R}$. In FP systems the precision value p refers to the length or number of bits of the significand, for example, the FP representation used by the Zuse Z1 contained one sign bit, seven exponent

bits and a fourteen bit significand, resulting in a precision $p = 14$ [149]. Number systems with a higher value for p will have a higher accuracy during computation as the system is able to represent a larger number of real numbers $x \in \mathbb{R}$.

2.4.1 Normalized Values

In practice a majority of FP formats will use a *normalized* significand [65, 131]. Using normalized values the significand is aligned so that the leading non-zero digit is stored immediately to the left of the radix point as the MSB followed by $p - 1$ digits after the radix point such that $1 \leq m < \beta$. This results in the following two properties:

1. Representable numbers $x \in \mathbb{F}$ have a *unique* floating point representation as the minimum value for e that is greater than or equal to e_{min} must be determined for a given value [131].

2. The precision of radix-2 (binary) systems can be extended by assuming a value for the MSB of the significand under specific conditions [95].

In the case of a binary normalized significand the MSB has an assumed value of one and is not stored. Using the above definitions a binary normalized FP number $x \in \mathbb{F}$ can be defined as:

$$x = (-1)^s \left[1 + \sum_{i=1}^{p-1} m_{-i} \beta^{-i} \right] \beta^{e_b} \tag{2.48}$$

where $s \in \{0, 1\}$ is the sign, $p \geqslant 2$ is the precision, $1 \leq 1.m < \beta$ is the normalized significand, $\beta = 2$ is the radix and $e_{min} \geq e_b \leq e_{max}$ is the biased exponent. Alternatively subnormal numbers will be indicated by $e = e_{min} - 1$ and will have a significand value $0 \leq m < 1$ and can be defined by the following:

$$x = (-1)^s \left[\sum_{i=1}^{p-1} m_{-i} \beta^{-i} \right] \beta^{e_{min}} \tag{2.49}$$

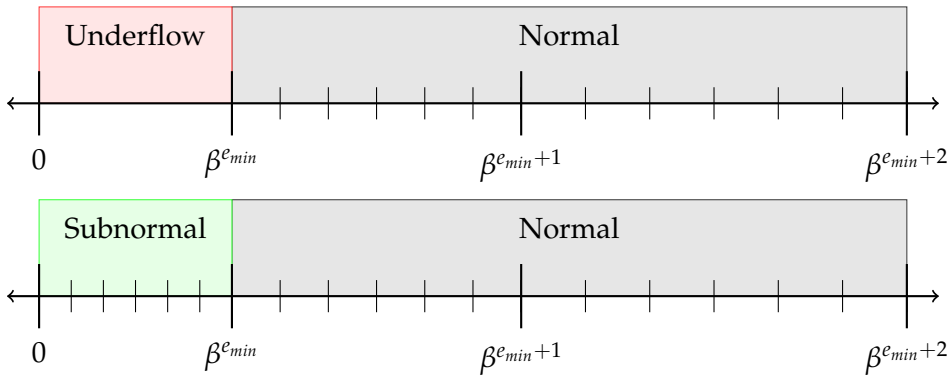


Figure 2.1: Normal & Subnormal Numbers

Note that although an exponent value $e = e_{min} - 1$ is used to represent subnormal values, the resulting value is calculated using $e = e_{min}$. Subnormal values are considered the most difficult type of FP value to implement and as such some implementations of FP number systems do not include subnormal values or alternatively will implement these values using software (SW) methods [95, 131]. The benefit of subnormal numbers is the ability to represent otherwise non-representable values between zero and the smallest normal value ($x = \beta^{e_{min}}$). In a system that contains only normalized values any value smaller than $\beta^{e_{min}}$ must be rounded, either to $\beta^{e_{min}}$ or alternatively flushed to zero, resulting in a loss of significance of p digits referred to as underflow, and a large gap between zero and $x = \beta^{e_{min}}$, (commonly referred to as the zero gap). Subnormal numbers fill the zero gap and allow for representation of these values using leading zeroes in the significand, use of subnormal representation is therefore known as gradual underflow [95]. Figure 2.1 shows how the range of possible values will differ within a system when subnormal numbers are used as opposed to a system that only uses normalized numbers.

2.4.2 Exact & Inexact Values

An important concept within FP arithmetic is the distinction between *Exact* and *Inexact* values. As FP arithmetic is a real-world application used by systems with finite performance and memory resources, the system is limited to a finite precision and therefore is not able to represent every possible value within the infinite set of

real numbers, $\mathbb{R}$. Instead the set of FP numbers $\mathbb{F}$ is a finite subset of real numbers $\mathbb{F} \subset \mathbb{R}$. Real numbers that are representable in FP format are referred to as exact values while inexact values refer to real numbers that cannot be represented and are instead rounded to the nearest exact value (nearest value determined by the rounding scheme) [53, 65, 131]. An inexact value can be thought of as a value that falls between two exact values, or, is a value that differs from an exact value by less than the unit in the last place (ULP). The ULP is a measurement of the smallest possible representable FP value available for a given exponent value, or more formally [131]:

The $ulp(x)$ is the gap between the two FP numbers nearest to x

The ULP is calculated by:

$$ulp(x) = \beta^{max(e,e_{min})-p+1} \tag{2.50}$$

The value ULP is often used in reference to FP error, which is the difference between a real value $x \in \mathbb{R}$ and its FP representation $\hat{x} \in \mathbb{F}$. In the case of an inexact value the maximum possible error will be one $ulp(\hat{x})$:

$$|\hat{x} - x| \leq ulp(\hat{x}) \tag{2.51}$$

The concept of ULP is closely tied to *approximation error*, a measure of the discrepancy between an exact value and its approximation, defined by both the absolute error:

$$\epsilon = |\hat{x} - x| \tag{2.52}$$

and the relative error, (assuming $x \neq 0$):

$$\delta = \frac{\epsilon}{|x|} \tag{2.53}$$

$$= \left| \frac{\hat{x} - x}{x} \right| \tag{2.54}$$

In the case of a correctly rounded FP format, the absolute and relative errors are limited as follows:

$$\epsilon \leq ulp(\hat{x}) \tag{2.55}$$

$$\delta \leq \beta^{-p} \tag{2.56}$$

2.4.3 Rounding

Due to the existence of inexact values methods must be made available to identify these values and convert them to values that can be represented by the system. This requires methods for rounding to be implemented as part of a FP system. Until the implementation of the Institute of Electrical and Electronic Engineers (IEEE) standard for FP arithmetic there was no available standard for FP rounding and methods for rounding varied from system to system. The implementation of the IEEE FP standard called for rounding of FP operations to be standardized by implementing a set of rounding modes that controlled both the precision and direction of rounding. In his original paper on the proposed IEEE standard [94], Kahan stated that FP rounding could be standardized according to a very simple model:

The rounded result will be one of the neighbours of the infinitely precise true result, depending on the direction of rounding

According to this model if the result of a FP operation is inexact, then the rounded result must be one of the two exact values closest to the un-rounded value. Stated more formally, the rounded approximation $\hat{x} \in \mathbb{F}$ of an exact value $x \in \mathbb{R}$ must be

within one ULP for single-direction rounding, (round up/down, round to zero or round to $\pm\infty$);

$$x - ulp(x) \leq \hat{x} \leq x + ulp(x) \tag{2.57}$$

Or within half a ULP for multi-direction rounding, (round to nearest even/odd);

$$x - \frac{1}{2}ulp(x) \leq \hat{x} \leq x + \frac{1}{2}ulp(x) \tag{2.58}$$

A full description of rounding methods implemented by the IEEE-754 standard is available in Section 2.5.3.

2.4.4 Overflow & Underflow

Overflow and underflow are exceptions that occur when the result is a value that is either too large or too small to be represented in FP format [65, 131]. Specifically:

1. Overflow will occur when the exponent of the result is larger than the maximum exponent of the system, $e > e_{max}$.

2. Underflow will occur when the exponent of the result is less than the minimum exponent of the system, $e < e_{min}$. In this case the result is typically flushed to zero.

When these exceptions occur they are handled by assigning special values to the results, usually values representing infinity and zero for overflow and underflow respectively, and setting exception flags to indicate the exceptions have occurred. The specific values assigned and their formats vary from system to system with standard formats and exception behaviour being defined for the IEEE-754 standard (detailed in section 2.5.2). The introduction of subnormal values allows for *gradual underflow* to occur. In this case a value that would normally underflow to zero will instead be rounded to the nearest subnormal value [94, 95].

Format	β	p	e_{min}	e_{max}	range	width
binary16	2	11	-14	15	$\approx 10^{\pm 5}$	16-bits
binary32	2	24	-126	127	$\approx 10^{\pm 38}$	32-bits
binary64	2	53	-1022	1023	$\approx 10^{\pm 308}$	64-bits
binary128	2	113	-16382	16383	$\approx 10^{\pm 4932}$	128-bits

Table 2.3: IEEE-754 Binary Formats.

2.5 IEEE-754 Floating Point Standard

In 1979 Kahan proposed a standard implementation of FP arithmetic [95] that eventually became the first IEEE-754 standard. The standard now has three versions, the first, the IEEE-754:1985 standard [50] implemented binary FP arithmetic only and in 1987 was augmented with IEEE-854:1987, which implemented both binary and decimal formats of FP arithmetic [51]. The current standard is the IEEE-754:2008 standard which implements three decimal and four binary formats and is considered the default standard for performing floating point arithmetic [52]. The basic parameters for each binary format are provided in table 2.3. Binary IEEE-754 numbers $X \in \mathbb{F}(\beta, p, e_{min}, e_{max})$ implemented by the system can be either normal or subnormal values. Normalized values are of the form:

$$x = (-1)^s \left[1 + \sum_{i=1}^{p-1} m_{-i} 2^{-i} \right] 2^{e_b} \tag{2.59}$$

and subnormal values take the form:

$$x = (-1)^s \left[\sum_{i=1}^{p-1} m_{-i} 2^{-i} \right] 2^{e_{min}} \tag{2.60}$$

For all cases $s \in \{0, 1\}$ is the sign, and $p \geq 2$ is the precision of the system. In the case of normal numbers $1 \leq m < 2$ is the significand, e is the exponent and $e_b = e - e_o$ is the biased exponent where $e_{min} \leq e_b \leq e_{max}$. Subnormal numbers are indicated by a biased exponent value $e_b = e_{min} - 1$ and significand $0 \leq m < 1$. For all cases the sign, exponent and significand values are stored in the format shown in figure 2.2.

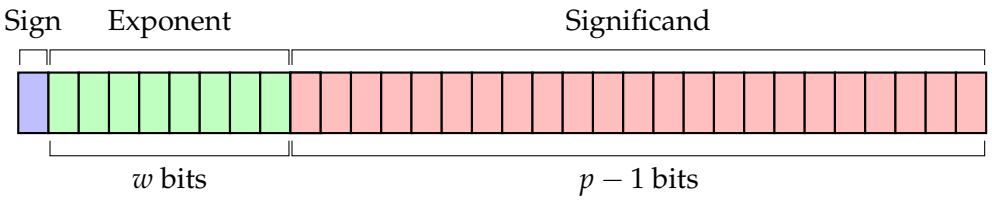


Figure 2.2: IEEE-754 Encoding Format

2.5.1 Basic Operators

The IEEE-754 standard defines a number of operators that can be used for FP arithmetic, including definitions for operations such as comparison and conversion, however this section will focus on the arithmetic operations listed in the standard, add, subtract, multiply, divide, fused multiply add (FMA) and square root. This following section outlines the basic procedure for performing these operations using FP arithmetic. In the case of either addition or subtraction the operation is performed according to the following formula;

$$r = x \pm y \tag{2.61}$$

$$= \left[(-1)^{s_x} m_x \pm (-1)^{s_y} \left(\frac{m_y}{2^{e_x - e_y}}\right)\right] 2^{e_x} \tag{2.62}$$

which can be implemented in the following steps:

1. Align significand values: If the exponent values e_x and e_y are not equal then the significand must be aligned before the fixed point addition operation can be performed. This is done by calculating the difference of the exponent values $e_x - e_y$ then shifting the value m_y by this value:

$$m_y = m_y \gg e_x - e_y \tag{2.63}$$

2. Add significand values: The value of the significand of the result m_r is calculated using fixed point addition methods. The type of FP operation begin performed and the value of the input sign bits determines whether the signi-

Operation	S_x	S_y	Result
Add	0	0	$m_r = m_x + m_y$
Add	0	1	$m_r = m_x - m_y$
Add	1	0	$m_r = m_y - m_x$
Add	1	1	$m_r = -(m_x + m_y)$
Subtract	0	0	$m_r = m_x - m_y$
Subtract	0	1	$m_r = m_x + m_y$
Subtract	1	0	$m_r = -(m_x + m_y)$
Subtract	1	1	$m_r = m_y - m_x$

Table 2.4: Fixed point operations on significand values for addition/subtraction

ficand values are summed or subtracted, the truth table shown in Table 2.4 details the boolean logic required to determine the fixed point operation used.

$$m_r = m_x \pm m_y \tag{2.64}$$

If the m_r is negative s_r is set to one and the two's complement of m_r found.

3. Normalize: The number of leading zeroes λ_r of m_r is determined and m_r shifted by this amount. The exponent value e_r is also calculated at this point. If the leading zero is behind the radix point λ_r is negative (left shift), and if the leading zero is to the left of the radix point (either the MSB or the carry bit are high) λ_r is positive (right shift).

$$m_r = m_r \gg \lambda_r \tag{2.65}$$

$$e_r = e_r + \lambda_r \tag{2.66}$$

4. Perform rounding and renormalise if necessary.

Multiplication and division operations are performed as follows:

$$r = x \cdot y \tag{2.67}$$

$$= (-1)^{s_x \oplus s_y}(m_x \cdot m_y)2^{(e_x+e_y)} \tag{2.68}$$

$$r = x/y \tag{2.69}$$

$$= (-1)^{s_x \oplus s_y}\left(\frac{m_x}{m_y}\right)2^{(e_x-e_y)} \tag{2.70}$$

both multiplication and division can be implemented using the following steps:

1. Calculate the sign value of the result s_r:

$$s_r = s_x \oplus s_y \tag{2.71}$$

2. Calculate the value of the significand m_r using fixed point arithmetic:

$$m_r = \begin{cases} m_x \cdot m_y & \text{multiply} \\ m_x/m_y & \text{divide} \end{cases} \tag{2.72}$$

   The exponent value e_r is also calculated at this point using fixed point arithmetic:

$$e_r = \begin{cases} e_x + e_y & \text{multiply} \\ e_x - e_y & \text{divide} \end{cases} \tag{2.73}$$

3. Normalize: the number of leading zeroes λ_r is determined and the significand of the result shifted into the correct position, the exponent of the result is also adjusted:

$$m_r = m_r \gg \lambda_r \tag{2.74}$$

$$e_r = e_r + \lambda_r \tag{2.75}$$

4. Perform rounding and renormalise if necessary.

The FMA operation implements a multiply and accumulate operation with a single rounding stage. Using the equations for addition, $r = x + y$, and multiplication, $r = x \cdot y$, given previously the FMA operation $r = x + (y \cdot z)$ can be defined by substituting the equation for multiplication into the second term of the equation for addition;

$$r = x + (y \cdot z) \tag{2.76}$$

$$= \left[(-1)^{s_x} \cdot m_x + (-1)^{s_y \oplus s_z} \cdot \frac{m_y \cdot m_z}{2^{e_x - (e_y + e_z)}}\right] \cdot 2^{e_x} \tag{2.77}$$

which can be implemented as follows;

1. As in the case of multiplication, calculate the sign, significand and exponent of the result of the multiply stage:

$$s_m = s_y \oplus s_z \tag{2.78}$$

$$e_m = e_y + e_z \tag{2.79}$$

$$m_m = m_y \cdot m_z \tag{2.80}$$

2. Instead of performing normalization and rounding the results of the multiplication operation are now used for the addition stage. As in the case of standard addition, the significands must first be aligned:

$$m_y \cdot m_z = (m_y \cdot m_z) \ll e_x - (e_y + e_z) \tag{2.81}$$

$$m_m = m_m \ll e_x - e_m \tag{2.82}$$

3. Determine the value of the result significand, m_r, using fixed point addition

methods with the sign and operation determined as in standard addition:

$$m_r = m_x \pm (m_y \cdot m_z) \tag{2.83}$$

$$= m_x \pm m_m \tag{2.84}$$

4. Normalize: The number of leading zeroes, λ_r, in the result significand is determined, and the result exponent and significand are adjusted as follows:

$$m_r = m_r \gg \lambda_r \tag{2.85}$$

$$e_r = e_r + \lambda_r \tag{2.86}$$

5. Perform rounding and re-normalize if necessary.

The final operation detailed in this section is the FP square root operation $x = \sqrt{a}$, where a is a positive real number; $\{a \in \mathbb{R} : a \geq 0\}$. Typically this operation is implemented using iterative techniques, one of the most common being the Newton-Raphson Method. As in the case for fixed point division an algorithm will implement the general form of the Newton-Raphson method as shown in Equation 2.37 resulting in two possible derivations of the algorithm. The naive implementation utilizes the simplest form of the base function, $f(x) = x^2 - a$, resulting in the following expansion of the general form;

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} \tag{2.87}$$

$$= \frac{1}{2} \cdot \left(x_n + \frac{a}{x_n} \right) \tag{2.88}$$

In the general case Newton-Raphson provides a fast method for computing square roots, and using this implementation the iterative result, x_n will converge to $\sqrt{a}$ in $O(\log(p))$ iterations assuming the initial estimate is greater than zero $x_0 > 0$, however this implementation requires a divide operation be performed at each stage. As the division operation is expensive in and of itself the naive implementation is

not ideal and can be improved by modifying the base function so that the resulting algorithm determines the reciprocal square root, $x = \frac{1}{\sqrt{a}}$. Once the reciprocal is determined the square root may be calculated using a single multiplication operation, $\sqrt{a} = a \cdot \left(\frac{1}{\sqrt{a}}\right)$. Using this method the base function is set to $f(x) = \frac{1}{x^2} - a$ resulting in the following expansion of the general form:

$$x_{n+1} = x_n - \frac{x_n^{-2} - a}{-2x^{-3}} \tag{2.89}$$

$$= \frac{x_n}{2} \cdot \left(3 - ax_n^2\right) \tag{2.90}$$

Although this version also contains a division operation, it is division by two which can be performed using a right shift operation in binary systems, or by multiplying by one half. Further performance improvements can be made by using a FMA operation to perform the calculation of $3 - ax_n^2$. Using this implementation the system will converge to the solution $\frac{1}{\sqrt{a}}$ in quadratic time, however this requires the initial estimate x_0 to be a close approximation of the final solution, obtainable using a look-up table or polynomial approximation techniques. The Newton-Raphson reciprocal square root method is also used for the implementation of the Fast Inverse Square Root method, an algorithm originally developed for 3D graphics processing in game development and first appearing in the source code for Quake III [117]. Using this method the performance of the Newton-Raphson method is improved by determining a better approximation of the solution for the initial value, x_0, using what is now known as the *Magic Number* [117]. An alternative to the Newton-Raphson method is the Goldschmidt algorithm for simultaneous calculation of the square root and the reciprocal square root [120]. Using Goldschmidt's algorithm to find the square root and reciprocal square root, $\sqrt{a}$ and $\frac{1}{\sqrt{a}}$ an initial estimate is

calculated;

$$b_0 = a \tag{2.91}$$

$$Y_0 \approx \sqrt{a} \tag{2.92}$$

$$y_0 = Y_0 \tag{2.93}$$

$$x_0 = a \cdot y_0 \tag{2.94}$$

The approximate estimate for the value of Y_0 is typically determined using a lookup table. Having determined the initial estimates the iterative method is performed as follows;

$$b_{i+1} = b_i \cdot Y_i^2 \tag{2.95}$$

$$Y_{i+1} = \frac{1}{2}(3 - b_i) \tag{2.96}$$

$$y_{i+1} = y_i \cdot Y_{i+1} \tag{2.97}$$

$$x_{i+1} = x_i \cdot Y_{i+1} \tag{2.98}$$

until the value b_i converges to 1 or a maximum number of iterations is reached, the values of the square root and reciprocal square root are found from x_i and y_i respectively:

$$\sqrt{a} = \lim_{i \to \infty} x_i \tag{2.99}$$

$$\frac{1}{\sqrt{a}} = \lim_{i \to \infty} y_i \tag{2.100}$$

2.5.2 Special Values

The standard defines a set of values that are used when the result of a computation cannot be formatted as either a normal or subnormal value. The special values will normally result from an exception and ideally will not form the result of a standard computation. The values fall into three basic categories; zeroes, infinities and not a number (NaN). The types of special value and their basic formats are shown

Type	Value	s	e	m
Positive Zero	$+0$	0	$e_{min}-1$	0
Negative Zero	-0	1	$e_{min}-1$	0
Positive Infinite	$+\infty$	0	$e_{max}+1$	0
Negative Infinite	$-\infty$	1	$e_{max}+1$	0
Not a Number	NaN	0	$e_{max}+1$	>0

Table 2.5: IEEE-754 Special Value Formats.

in table 2.5. The NaN type contains two possible NaN formats. The quiet not a number (qNaN) and the signaling not a number (sNaN). A sNaN value will not result from an arithmetic operation but instead will trigger an invalid operation exception when used as an operand, alternatively a qNaN value can propagate through an operation and be returned as a result whenever an invalid operation occurs. The MSB of the significand is used as a flag to indicate whether the NaN value is a qNaN or a sNaN.

2.5.3 Rounding

The IEEE-754 standard defines a set of rounding modes that can be used to map an exact value to the best exact approximation. The standard defines a set of five rounding modes with each mode containing rules for the direction and precision of the mode. The five are defined as follows:

1. **Round to $+\infty$:** Also referred to as round up, the rounded value of x will be the smallest possible FP value that is greater than or equal to x:

$$RU(x) \geq x \tag{2.101}$$

2. **Round to $-\infty$:** Also referred to as round down, the rounded value of x will be the largest possible FP value that is less than or equal to x:

$$RD(x) \leq x \tag{2.102}$$

3. **Round to** 0**:** The rounded value of x will be the nearest FP value that is less than or equal to x:

$$RZ(x) = \begin{cases} RD(x) & x > 0 \\ RU(x) & x < 0 \end{cases} \tag{2.103}$$

4. **Round to Nearest (odd):** The rounded value of x is the nearest FP value. If x is exactly halfway between two FP values, the result will be the nearest odd value.

5. **Round to Nearest (even):** The default rounding mode for IEEE-754. The rounded value of x will be the nearest FP value, or the nearest even FP value if x is exactly halfway between two FP values.

In practice rounding is implemented by appending a set *round bits* to the end of the significand during the operation, effectively extending the precision of the fixed point operations that make up the overall FP operation. For IEEE-754 a total of three rounding bits are used, referred to as the *guard, round* and *sticky* bits, which are appended to the significand in that order. During normalization or significand alignment any bits shifted off the right of the significand will be passed through the round bits. The sticky bit has a special function in that it will "stick" to a high value once a high bit is shifted to its position effectively acting as an indicator bit for any information lost during the shift. During rounding the round bits and the round mode are used to determine how a value will be rounded as shown in table 2.6. *The trunc() operation indicates that the round bits are simply truncated from the significand, effectively subtracting them from the final value.*

2.5.4 Exceptions

The IEEE-754:2008 standard defines a set of five exceptions that must be handled by a compliant system. Each exception defines both default exception handling and alternate handling, with each definition detailing default values and the status flags

Rounding Bits			Rounding Mode		
Guard	Round	Sticky	Round Down	Round Up	Round Nearest
X	0	0	$trunc(x)$	$trunc(x)$	$trunc(x)$
X	0	1	$trunc(x)$	$x + 2^{-p}$	$trunc(x)$
X	1	0	$trunc(x)$	$x + 2^{-p}$	$trunc(x)$ or $x + 2^{-p}$
X	1	1	$trunc(x)$	$x + 2^{-p}$	$x + 2^{-p}$

Table 2.6: IEEE-754 Rounding Operations.

that will be raised. The exceptions are designed so that an exception can be detected and handled without interrupting program execution. The five exceptions and their default behaviour are defined as follows [52]:

1. **Inexact:** If the result of the operation is inexact, that is the result differs from the result computed using infinite exponent range and precision, and inexact exception will occur. This will result in the inexact status flag being raised, the result returned will be the rounded result.

2. **Invalid Operation:** The invalid operation will occur when an operation occurs that has no definable result, such as and operation on a NaN, divide by zero or adding/multiplying infinities. When an invalid operation occurs the invalid status flag is raised and the result will be set to a quiet NaN value providing diagnostic information.

3. **Divide by Zero:** The divide by zero exception will only occur if an infinite result is defined for finite operands. The exception results in the divide by zero status flag being set and the result being set to infinity. The sign of the result is set depending on the sign of the operands.

4. **Overflow:** An overflow exception will occur when the magnitude of the rounded result is larger then the magnitude of the formats largest finite number. In the case of an overflow exception the overflow status flag is set, the result of the operation will depend on the rounding mode and the sign of the operand:

   (a) Round to Nearest (Even/Odd) - All results set to infinity with the sign of

the intermediate result.

(b) Round to Zero - All results set to the largest finite result with the sign of the intermediate result.

(c) Round Up - Negative results set to the largest finite result with negative sign, positive results set to positive infinity.

(d) Round Down - Positive results set to the largest finite result with positive sign, negative results set to negative infinity.

5. **Underflow:** An underflow exception will occur when the result is a subnormal value, that is when the result that would be computed given infinite exponent range and precision falls within the range $-\beta^{e_{min}} \leq x \leq \beta^{e_{min}}$. The default handling of an underflow exception will deliver a rounded result. If the result is inexact both the inexact and the underflow exception will be raised, if the result is exact, (exact underflow, no rounding required), no flag is raised.

2.6 Error Analysis

Most number systems are limited to a finite precision due to the limits on memory and performance that exist in all real world computer systems. This limitation on precision means that not all real numbers are representable, in fact the gap between any two FP values will contain an infinite set of real numbers that cannot be represented. In practice the FP number system provides the best available approximation of a number system for use in computer arithmetic and will usually provide results that are accurate enough for the task at hand, however, certain factors such as errors of measurement or estimation, quantization error or errors propagated from earlier parts of a computation can result in inaccurate results. Several systems have been developed for the detection and analyses of errors FP computations, including interval arithmetic (IA), affine arithmetic (AA) and Monte Carlo arithmetic (MCA).

This dissertation focuses on errors that can occur in FP arithmetic, (specifically

the IEEE-754:2008 standard), as these are the most common types of error due to the widespread implementation of FP. The following section provides a description of the types of arithmetic error that are most likely to affect a FP computation, the factors that can lead to these types of error, the effects that these errors can have on a computation and the methods used to measure and analyse arithmetic error and stability.

2.6.1 Arithmetic Error

Errors resulting from FP computations are often overlooked, due to the fact that the error is either too small to notice, appeared then disappeared too quickly to be noticed, or due to the simple fact that the computation was not important enough to warrant error analysis [94]. The widespread implementation of FP systems means that errors with serious consequences can occur, and have occurred in the past. Some of the more infamous examples include Intel's floating point division (FDIV) bug, which caused errors in FP division operations and cost the company hundreds of millions of dollars in recall costs [65, 140], and the explosion of a 7 billion dollar Ariane 5 unmanned rocket launched by the European Space Agency, an explosion caused by errors in the conversion of a 64-bit FP value to a 16-bit signed integer [105, 140]. Errors due to round-off and precision are not limited to FP arithmetic. In 1991 during the first Gulf War a rounding fault existed in the fixed point system used for guidance in the Patriot Missile system. This fault eventually led to the miscalculation of the trajectory of an incoming Scud Missile that killed 28 U.S soldiers at a base in Dahran, Saudi Arabia [140, 158].

2.6.2 Numerical Stability

The property of numeric stability is a desirable property of any algorithm or arithmetic system. A measure of numeric stability is essentially a measure of the accuracy of a given algorithm, in many cases a problem can be evaluated in one of several different ways each performing the same task but with the possibility that each

method will yield different results due to finite precision, rounding or quantization errors. Numeric stability is used to determine the level to which the results of an implementation agree with each other or with an ideal result, and can be applied to a number system, an individual representation of a real value or to an algorithm.

Two of the most basic measures of numeric stability are relative error and significant figures. As explained in Section 2.4.2 the relative error of a value measures the difference between the representation X, and the corresponding real value (calculated using infinite precision) x [65, 82, 131]:

$$\delta = \frac{|x - X|}{x} \tag{2.104}$$

Significant figures are defined as the figures of a number which give meaning to its precision and are defined as the first non-zero digit and subsequent trailing digits (including zeroes), i.e. in a five digit decimal number system with two whole and three fraction digits, the value 00.005 has **one** significant figure while the value 01.050 has **four** significant figures. Measurement of stability using significant figures is preformed by determining to what level the significant figures agree, two algorithms attempting to solve the same problem may produce the values 13.051 and 13.052, these results can be said to agree to four significant figures. Two values may agree to a high number of significant figures while still having substantially different relative errors, making one answer more accurate than the other despite the agreement in terms of significant figures. For this reason relative error and significant figures are often combined when used to measure the stability of an algorithm [82, 140].

A more formal definition of numeric stability is separated into definitions for forward, backward and mixed stability. Using these definitions the errors affecting a function $f(x)$ can be separated into these three categories and analysed appropriately. If the function $y = f(x)$ is a function for mapping data x to the solution

y then the result of the function as performed using some finite precision system will be an approximation of the result, $\hat{y}$. Computing the result of this function will also involve converting the value x to its finite precision representation, $\hat{x}$. Both operations will result in the possibility of error begin included in the result. The conversion of the value x is modelled by $\hat{x} = x(1+\delta)$ and the calculation of the final result is modelled by $\hat{y} = f(x(1+\delta))$ [82, 140]. The error value in $\hat{x}$ represents errors in measurement, quantization or errors propagated forward from earlier calculations and is referred to as the *forward error* of the system, alternatively the relative error in $\hat{y}$ represents errors due to rounding and is referred to as *backward error*. Measurement and analysis of the backward error of an algorithm is referred as backward error analysis and is used to determine if a function is backward stable and determine its sensitivity to input perturbations. A function can be said to be backward stable if the function produces a correct value $\hat{y} = x(1+\delta)$ for small perturbations of δ, i.e. the function is not affected by small changes to the forward error of the inputs [82, 140].

2.6.3 Round-off Error

FP arithmetic can be defined as a type of rounded arithmetic, that is, not all possible possible real values $\mathbb{R}$ can be represented and instead are approximated by FP values $\mathbb{F}$. This leads to the concept of exact and inexact values as discussed in section 2.4.2. A more formal definition of this concept given in [64, 82, 140, 142] states that the accuracy of the rounded approximation $\mathbb{F}(x)$ of a value x is given by:

$$\mathbb{F}(x) = x(1+\delta) \tag{2.105}$$

$$\delta \leq \epsilon \tag{2.106}$$

Where $\epsilon = \frac{1}{2}\beta^{1-p}$ is the machine epsilon of the system. From this equation it can be seen that all inexact values will contain an error in their approximation. Round-off error can be defined as a type of backward error, as it is introduced during an operation, however, round-off error will normally include errors taken from the

inputs to the operation, and as such a function including both rounding (backward) error and forward error can be defined as follows [64, 82, 140]:

$$\hat{y} + \Delta = f(x(1+\delta)) \tag{2.107}$$

where $\Delta \leq ulp(\hat{y})$ and $\delta \leq ulp(\hat{x})$. In this model the final value for the result contains values for both the forward and backward error to represent the rounding error in the operation, a situation referred to as mixed forward-backward error [82].

2.6.4 Catastrophic Cancellation

Cancellation is an phenomenon that will occur when two nearly equal values are subtracted leaving a large number of zeroes after the radix point, in floating point arithmetic this situation cannot occur when dealing with normalized values and as a result catastrophic cancellation can occur. If one or more non-exact numbers are subtracted, a loss of significant digits can occur due to normalization of the result [64, 65, 82, 93]. This phenomena is called *Catastrophic Cancellation* and is one of the major causes of loss of significance. Consider the solution to the equation

$$x^2 + 444x + 1 = 0 \tag{2.108}$$

using the quadratic formula

$$r = \frac{\left(-b \pm \sqrt{b^2 - 4ac}\right)}{2a} \tag{2.109}$$

IEEE-754 single precision format uses a 24-bit binary significand giving it a precision value $p = 24$, equivalent to $log_{10}(2^{24}) \approx 7.225$ decimal digits. In most cases the

answer will be accurate to 7 decimal places, but in this example the exact result is:

$$r_1 = -222 \pm \sqrt{49283} \tag{2.110}$$

$$= -0.00225226368 \tag{2.111}$$

whereas IEEE-754 arithmetic gives $r_1 = 0.000000000$. This has a 100% relative error due to catastrophic cancellation. A better insight to the effects of catastrophic cancellation can be seen by considering the equation $\hat{x} = \hat{a} - \hat{b}$ where $\hat{a} = a(a + \delta_a)$ and $\hat{b} = b(1 + \delta_b)$, in this situation the relative error of the function is given by [82]:

$$\left| \frac{x - \hat{x}}{x} \right| = \left| \frac{-a\delta_a - b\delta_b}{a - b} \right| \leq max(|\delta_a|, |\delta_b|) \frac{|a| + |b|}{a - b} \tag{2.112}$$

This shows that the relative error is large when $|a - b| \ll |a| + |b|$, i.e. when catastrophic cancellation occurs it will magnify errors already present in the operands.

2.6.5 Static & Dynamic Error Analysis Methods

The design of numerically stable algorithms must ensure that the issues reviewed do not adversely contribute to the accuracy of the solution. In the design of numeric libraries, analysts use techniques such as forward and backward error analysis to quantify the propagation of errors and understand their effect on the stability and accuracy of the algorithms [170]. Unfortunately, these techniques cannot be applied to arbitrary programs, require manual analysis and considerable expertise, and do not scale beyond small subroutines.

One of the primary questions in the study of numeric analysis is not how to develop the best techniques or systems, but how to get the best techniques and systems into the hands of the developers working with real world problems. Aside from the practical considerations of ease of understanding, implementation and use, there exists the question of what developers need or want in a numeric analysis tool. One

of the best assessments of numeric analysis techniques and the current state of the art is presented in [96]. Kahan notes a significant problem in encouraging the adoption of numeric analysis techniques; the average developer is not interested in these techniques until **after** something has gone wrong, at which point analysis is often required for *"...an assignment of blame and the task of relieving the distress, if possible."* It is for this reason, among others, that developers often search for what Kahan calls **mindless** assessments of round-off error, essentially systems that allow for a **fire and forget** approach rather than an in-depth analysis of the inner numeric workings of a piece of software. When viewed through this lens, the question of how to design systems that will be eagerly adopted by developers becomes a philosophical difference between two approaches to numeric analysis:

- How many significant digits are available in the results, or, how accurate is my program?
- What is the worst case bound on the absolute/relative error, or, how badly could my program fail?

What Kahan refers to as mindless assessments of round-off error often focus on the second approach, as this is the question that developers want answered after something has gone wrong (in which case the question often becomes how badly **did** my program fail?). The remainder of this chapter presents an overview of error analysis methods and the current state of the art.

Error analysis methods for software are divided into two types, **dynamic**, which analyses the results of program execution for a specific input set, and **static**, which is performed without the need for execution. While these analysis types are intended to be complimentary and may be used to validate each others' results, key differences exist. Dynamic analysis provides a higher level of flexibility and can even be performed without access to the source code in the case of automated tools, but more often requires significant modifications to the source code, and due to its

data dependency must be performed using an adequate set of inputs to produce meaningful results. Conversely, by limiting analysis to individual executions of a system, dynamic analysis methods are efficient, as system properties need only be checked along a single execution path. Furthermore, testing is conducted using actual operations performed by the system rather than mathematical abstractions allowing for more precise analysis. This also avoids compatibility issues being introduced from differences in arithmetic format, compilers or system architecture [125]. Static analysis avoids the data dependency issue by abstracting the possible states and operators of tested software, leading to a mathematical formulation that allows all possible states of a system to be tested. An overly rigorous definition will result in a complex analysis that does not scale to large systems. Automated tools for static analysis provide the ability to pinpoint the exact locations of errors in software, often at an earlier stage in the software development life cycle (SDLC), however automated tools only support certain languages and static analysis becomes time-consuming when performed manually.

Static analysis techniques typically use formal methods, whereby software is analysed using mathematical techniques based on formal semantics of the programming language used. These techniques include denotational semantics, axiomatic semantics, operational semantics and abstract interpretation. Methods used for static analysis include three basic types:

1. Model or Property Checking
2. Data Flow Analysis
3. Abstract Interpretation

Model or property checking requires the creation of formal models for both the system and its specification. Model checking may then be used to determine if the system model meets all requirements of the specification model [104]. In order to perform model checking algorithmically, it is limited to finite state systems and is

typically used for the analysis of hardware (HW) systems as the undecidability of SW limits its effectiveness. Due to this limitation model checking is often used for analysis of SW and HW systems modelled as a finite state machine (FSM). Data flow analysis is a technique for generating possible sets of values for nodes in a program's control flow graph (CFG), this is typically accomplished using an iterative approach that determines values for the in-states and out-states at each node in the CFG until the complete system stabilizes [29, 100]. Finally abstract interpretation creates partial abstractions of operations and variables in order to create a computable semantic interpretation. It is viewed as a partial execution technique for static analysis [30, 31]. The semantics created for abstract interpretation are defined as monotonic functions that relate elements of the system across ordered sets.

Systems available for static analysis of rounding errors include Fluctuat [68], Astree [12] and Polyspace [41]. Fluctuat performs abstract interpretation using an abstract domain based on AA for analysis of FP error. This tool is now being used by Airbus to automate accuracy analysis of control software [38]. Astree is based on IA methods and is designed for safety critical analysis, including FP error analysis [12]. This software is also being used by Airbus for automated software analysis [39]. Polyspace is used to locate potential run-time errors including arithmetic overflow, divide by zero and buffer overrun, the software is now supplied by MathWorks and is used in several industrial applications.

Several systems have been developed for performing dynamic analysis of FP SW. IA [127, 138] is a system for producing error bounds on rounding and measurement errors of an algorithm, as opposed to an exact result, this allows numerical methods to be developed that produce reliable result bounds for systems that would otherwise produce inconsistent results. Using IA an input value is defined as a range of

real values, rather than a single value [78, 79, 127]:

$$x = [a, b] \tag{2.113}$$

$$= \{x \in \mathbb{R} : a \leq x \leq b\} \tag{2.114}$$

The use of an interval as opposed to a single value removes issues with inexact values and round-off error. Rather than attempting to find the nearest approximation of an inexact value, an interval consisting of two exact values that can be said to contain the inexact value is found. The use of intervals of this type requires redefinition of basic operators. Using IA where the interval of a value $x \in \mathbb{R} = [a, b]$ and $y \in \mathbb{R} = [c, d]$ the following basic operators are defined [19, 127, 162]:

$$x + y = [a + c, b + d] \tag{2.115}$$

$$x - y = [a - d, b - c] \tag{2.116}$$

$$x \cdot y = [min(a \cdot c, a \cdot d, b \cdot c, b \cdot d), max(a \cdot c, a \cdot d, b \cdot c, b \cdot d)] \tag{2.117}$$

$$\frac{x}{y} = \left[min\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right), max\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right)\right] \tag{2.118}$$

The use of IA requires the selection of appropriate error bounds in order for the system to produce usable results. If the error bounds selected are too narrow, then the final interval could be a range that does not contain all possible results of the system, alternatively if the error bounds selected are too wide then the final interval result will be overly pessimistic and essentially unusable. In the case of FP arithmetic intervals are selected in order to include error bounds that account for rounding error and finite precision in the operands [19, 183]. If the function $\mathbb{F}(x)$ is defined as a conversion function designed to convert a real value x to a FP representation $\hat{x}$ then the result will be an exact FP value plus a possible error:

$$x = \mathbb{F}(x) + \delta \tag{2.119}$$

$$= \hat{x} + \delta \tag{2.120}$$

Where $0 \leq \delta \leq ulp(\hat{x})$ is the round-off error of the conversion function. Using IA this conversion function can instead produce an interval that represents the round-off error of the function and contains the inexact value x within the error bounds of the interval [19, 146]:

$$x = \hat{x} + \delta \tag{2.121}$$

$$= [x|\hat{x} - ulp(\hat{x}), x|\hat{x} + ulp(\hat{x})] \tag{2.122}$$

$$= [RD(x), RU(x)] \tag{2.123}$$

where $RD(x)$ and $RU(x)$ are the round down and round up operations. Using this logic it is assumed that if x is an inexact value then the interval of x will b the two exact FP values nearest to x. Alternatively if the value is exact then a degenerative interval will occur. One significant drawback of IA is operations that result in pessimistic, or overly wide error bounds, as an interval is designed to bound all possible outcomes of an operation an ideal result will be the maximum and minimum possible exact values, an overly wide interval will contain values that are not possible and indicate a higher level of instability within the operation [96, 183]. This effect will often occur due to a dependency issue within IA. Variables used within an interval operation are assumed to vary independently of one another over the full range of the interval, however this may not always be the case, if there are any constraints between the given intervals then not all available results within the interval range will be valid. If this issue occurs then the resulting interval will be much wider than expected. Several schemes have been developed to try and avoid these types of errors. One such method is to combine interval arithmetic with rounded arithmetic during fixed point calculations of significand values. A second method is to redefine the basic operations to perform inner interval arithmetic (IIA). Using IIA results with tighter error bounds can be obtained, the system redefines the basic operators

as follows:

$$[x_1, x_2] + [y_1, y_2] = [x_1 + y_2, x_2 + y_1] \tag{2.124}$$

$$[x_1, x_2] - [y_1, y_2] = [x_1 - y_1, x_2 - y_2] \tag{2.125}$$

$$[x_1, x_2] \cdot [y_1, y_2] = \begin{cases} [min(x) \cdot max(y), max(x) \cdot min(y)] : 0 \notin [x_1, x_2], 0 \notin [y_1, y_2] \\ max(y) \cdot [x_1, x_2] : 0 \in [x_1, x_2], 0 \notin [y_1, y_2] \\ [max(x_1 y_2, x_2 y_1), min(x_1 y_1, x_2 y_2)] : 0 \in [x_1, x_2], 0 \in [y_1, y_2] \end{cases} \tag{2.126}$$

$$\frac{[x_1, x_2]}{[y_1, y_2]} = \begin{cases} \left[\left(\frac{min(x)}{min(y)} \right), \left(\frac{max(x)}{max(y)} \right) \right] : 0 \notin [x_1, x_2], 0 \notin [y_1, y_2] \\ \left(\frac{1}{max(y)} \right) \cdot [x_1, x_2] : 0 \in [x_1, y_2], 0 \notin [y_1, y_2] \end{cases} \tag{2.127}$$

where $max(x) = max(|x_1|, |x_2|)$ and $min(x) = min(|x_1|, |x_2|)$ [183]. The equations for IIA solve the dependency issue of standard IA by treating equal intervals as the same variable and will result in degenerative intervals in the situations mentioned previously. Although this system will result in tighter error bounds if the system is not used carefully it can have the opposite problem to IA, the resulting error bounds will be too narrow and not contain all possible results of the system being tested, i.e. in certain situations an IIA system will produce results that do not have *guaranteed enclosure*. A solution to this issue proposes a system of random interval arithmetic (RIA) [183], where an operation will be evaluated using either standard IA or IIA with a random variable determining which system is used for an individual operation. Using RIA standard IA operations can be combined with IIA result in tighter error bounds if the operands being used are monotonic, that is if operands x and y are related by $x \geq y$ then the results of the operation must adhere to the same relationship $f(x) \geq f(y)$. Using this system operations are repeated analysed and results treated statistically. Results for the average center and range of the resulting intervals are obtained, then the average and standard deviation used to determine an approximate interval for the system [183].

AA [6, 36, 86] is an extension of IA that is designed to eliminate dependency problems within IA. The key difference between IA and AA is the representation of values. Rather than using intervals, AA represents values in affine form, $\hat{x}$, which is represented as a first order polynomial:

$$\hat{x} = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + x_3\epsilon_3 + ... + x_n\epsilon_n \tag{2.128}$$

where the values x_i are real coefficients and the values ϵ_i are unknown variables in the range $[-1, 1]$. The real coefficient x_0 is referred to as the central value while x_i values are referred to as partial deviations, the values ϵ_i are referred to as noise symbols. The affine form can be converted to interval form using the following equations:

$$\hat{x} = [x + \xi, x - \xi] \tag{2.129}$$

$$\xi = \sum_{i=1}^{n} |x_i| \tag{2.130}$$

The affine form of a value allows for not only the interval of a value to be stored but also relationships to other values, the noise symbols are used to represent error within the values, with each noise symbol representing a different source of error and the partial deviation values determining what level each noise symbol affects the value. As relationships to other variables are stored the dependency issue that normally affects interval based operations can be eliminated, however, AA operations are more complex in terms of performance requirements and storage.

Using AA basic operators are defined as follows [48]:

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + (x_1 \pm y_1)\epsilon_1 + ... + (x_n \pm y_n)\epsilon_n \tag{2.131}$$

$$\hat{x} \pm \alpha = (x_0 \pm \alpha) + x_1\epsilon_1 + ... + x_n\epsilon_n \tag{2.132}$$

$$\hat{x}\alpha = x_0\alpha + x_1\epsilon_1\alpha + ... + x_n\epsilon_n\alpha \tag{2.133}$$

where α is a real value $\alpha \in \mathbb{R}$. Multiplication is a more difficult operation to implement but can be represented by the following:

$$\hat{x} \cdot \hat{y} = \left(x_0 + \sum_{i=1}^{n} x_i\epsilon_i\right)\left(y_0 + \sum_{i=1}^{n} y_i\epsilon_i\right) \tag{2.134}$$

$$= x_0y_0 + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\epsilon_i + \left(\sum_{i=1}^{n} x_i\epsilon_i\right)\left(\sum_{i=1}^{n} y_i\epsilon_i\right) \tag{2.135}$$

Implementation of AA requires increase levels of memory and performance resources to complete operations. Implementations will store affine forms using a combination of BCD and FP variables to represent individual values, typically an affine form using n noise symbols will store the value x_0 in FP format, the value n in integer form, followed by n sets of a FP value x_i, (the partial derivation) and an integer i representing which noise symbol ϵ_i to use at that point. This storage format therefore requires $2n + 2$ words to store an affine form to n noise symbols [116].

Several publications have implemented systems for modelling FP error using AA using the following specialized case of an affine form to represent FP values and their associated errors [48, 86]:

$$\hat{x} = x_0 + x_1\epsilon_1 + max(|x|)\beta^{-q}\delta_1 \tag{2.136}$$

where $\epsilon_1 \in [-1, 1]$ is the variation symbol representing variations in the input, $\delta_1 \in [-1, 1]$ is the error symbol representing errors in rounding/quantization and $max(|x|)\beta^{-q}$ is the error bound the value x. Using this form operations are performed

using the following basic format:

$$\hat{r} = (x \circ y) + max(|z|)\beta^{-q}\delta \tag{2.137}$$

$$= r_0 + \sum v_i\epsilon_i + \sum w_i\delta_i \tag{2.138}$$

Using these affine forms errors in FP arithmetic can be tracked during execution and can provide more accurate estimations of the error bounds of FP systems than IA systems, however, as in IA the accuracy and efficiency of the system will decrease as the number of operations chained together increases [48]. AA is also more complex to implement as a hardware solution due to the increased requirements in terms of memory and performance, at present the majority of implementations are software based solutions.

The Contrôle et Estemation Stochastique des Aarondis Calculs (CESTAC) technique [165, 166] is an implementation of the probabilistic approach similar to MCA. Using CESTAC an execution is repeated N times with the rounding method of FP operations randomized by rounding the result up or down with 50% probability. Using this method the least significant bit of the result significand is perturbed at each arithmetic stage creating a set of N results R_N. As in MCA statistical analysis of the result set can be used to determine the accuracy of the algorithm used.

Several SW-based implementations of these methods have been published including control of accuracy and debugging for numerical applications (CADNA) [8, 89], an implementation of discrete stochastic arithmetic (DSA) that is based on the CESTAC method. An implementation of DSA also exists for the numerical validation of programs in arbitrary precision [70]. It uses the Multiple Precision Floating-Point Reliably (MPFR) library. Several SW libraries for IA are available including extensions for scientific computation (XSC), Gaol and a C++ template class available as part of the Boost library [15, 67, 101]. Sun Micro-systems has also provided support

for IA as part of their C/C++ compiler library [168]. IA has also been implemented as part of Gappa [34], a formal verification tool for fixed and FP arithmetic. Gappa utilizes forward error analysis in addition to IA and requires a bounded input in order to perform its analysis. Using this tool, bounds on the outputs of an algorithm are determined in addition to proofs on these bounds that may be checked via a proof assistant [131]. In order to maintain reasonable performance, a limited number of HW implementations of IA [4, 155, 161] and CESTAC [21] can be found in the literature. A SW implementation of MCA has been published by Parker [143] along with a set of test cases, however this implementation cannot be applied to existing source without significant modifications. A field programmable gate array (FPGA)-based implementation of MCA addition and multiplication with an area penalty of less than 22% over IEEE-754 was published by Yeung et. al. [181].

A separate class of analysis techniques have also been developed for bit width optimisation of arithmetic operators. While primarily aimed at fixed point implementations for DSP and FPGA systems, most are applicable to both fixed and FP formats. The multiple word length paradigm (MWLP) [26] is an analysis technique that uses perturbation and scaling analysis for fixed point arithmetic to perform error constrained word length optimization. The system uses user defined error constraints on signal to noise ratio (SNR) in order to optimize FPGA based DSP systems for area use, speed or power consumption. This system has been implemented for linear [27, 28] and non-linear [25] DSP systems and is the basis for Right-Size, a word-length optimization system for adaptive filters [24]. Bit width optimization methods have also been developed using static analysis techniques including AA and adaptive simulated annealing (ASA) [109]. These are designed to use range and precision analysis of fixed point implementations in order to guarantee the absolute error bounds of the system, and have been implemented in the tool MiniBit [110]. This type of analysis has also been expanded to the analysis of FP applications using Automatic Differentiation [59]. Mixed analysis methods for both fixed and FP sys-

tems have also been developed using mixed precision analysis for optimizing word lengths for speed, power consumption and area use in FPGA systems [60]. Mixed analysis tools are also available in the form of MiniBit+ [139], and the BitSize tool [61]. Finally FP analysis systems have been developed using profiling techniques based on tools such as Valgrind. Using these tools, FPGA-based arithmetic systems for DSP implementation may be optimized for speed, power consumption and area use. They perform mixed precision analysis of FP operations in order to identify operations that may be optimized by reducing the precision of the FP operations, or replacing FP operators with fixed point or dual fixed point operators [16]. This type of analysis has implemented as part of the FloatWatch tool [17].

2.7 Summary

FP arithmetic is one of the most widely implemented systems for computer arithmetic and represents the best solution for situations requiring flexibility in both precision and range. The industry standard implementation of FP arithmetic is the IEEE 754 implementation and implementations of this standard are widely used in a variety of applications mostly without issue. However, due to certain characteristics of the number system, namely issues with finite precision limitations and cancellation occurring during the normalization stage rounding errors are un-avoidable. In an ideal case the error in any result will be limited by the ULP or machine epsilon of the particular implementation in use, resulting in a bound on relative error limited by the precision of the FP format:

$$\delta \leq \beta^{-p} \tag{2.139}$$

It is often assumed that rounding error will be limited by this inequality, however, the rounding error can become significantly larger and in some cases many times larger than the original result creating relative error rates in excess of 100%. Of particular concern in this work is the issue of catastrophic cancellation, a phenomenon that will

occur when subtracting two similar values resulting in a high number of significant digits shifted off of the significand during the normalization stage of an operation. The scale and consequences of FP error is varied, and arguments have been made that given the widespread application of FP arithmetic the vast majority of errors appear and disappear without being noticed and without significant consequences. Yet major errors have occurred in critical applications and the consequences at times have been severe, the need for accurate and easy to use error analysis tools is readily apparent and the goal of numeric analysts should be the development of these tools as part of the SDLC. A number of numeric analysis tools have been developed, including methods for performing static analysis, dynamic analysis and bit-width optimization. In this work the focus is on dynamic methods for run-time probabilistic analysis of rounding errors and in particular the application of Monte Carlo Methods to analysis of floating point arithmetic, a method developed by D.S. Parker known as MCA. This analysis method is detailed in the next chapter.

Chapter 3

Monte Carlo Arithmetic (MCA)

3.1 Introduction

Monte Carlo arithmetic (MCA) is an application of the Monte Carlo method (MCM) to numerical analysis of floating point (FP) arithmetic. MCMs are a class of probabilistic algorithms used to obtain results for problems where it is difficult or impossible to solve the problem using deterministic methods. Using MCMs repeated simulations using random sampling are performed to obtain a distribution of results that may be analysed using statistical methods. The MCM was originally developed for use in particle physics experiments and is based on methods for statistical sampling [122–124]. MCA was originally developed by D.S. Parker [142] and is an extension of floating point arithmetic designed to simulate inexactness in floating point variables and operations using random perturbation of the input and output operands. This has the effect of turning error analysis of a program or algorithm into a statistical problem that can be analysed using standard statistical methods.

3.2 Monte Carlo Methods

3.2.1 History and Development

The development of the modern MCM has been made possible only in the 20th century with the advent of the digital computer. Previously, analysis methods of this type were known as Statistical Sampling. Statistical sampling methods have been in use for several centuries. One of the earliest problems in statistical sampling to be solved using integral geometry was posed by Georges-Louis Leclerc, Comte de Buffon, in 1777, now known as Buffon's Needles [23, 81]. The transition from statistical sampling to MCM occurred in 1945 at Los Alamos Laboratory during the Manhattan project. As part of the development of nuclear weapons, one of the earliest electronic computers, the electronic numerical integrator and computer (ENIAC) was developed. At this point in time Statistical Sampling methods were no longer in widespread use, primarily due to the large number of tedious calculations required, however, with the development of the computer, mathematicians John von Neumann, Stanislav Ulam and Nicholas Metropolis realized that statistical sampling methods could be re-invented and modernized using the computer to perform the required calculations. The original experiments devised using the new technique were intended so solve the problem of neutron diffusion in fissionable material. This class of problem involved assigning values to variables describing neutron position, velocity, impact position and impact type according to the probabilities assigned to each variable. The continued development of the MCM was a major driver behind the development of pseudo-random number generators, allowing for a significant increase in the efficiency and performance of the simulations conducted. Further development has lead to the creation of Markov chain Monte Carlo method (MCMCM) and quasi-Monte Carlo method (qMCM), with these techniques being used in a number of different fields.

3.2.2 Definition and Implementation

The definition of a MCM varies, but in the general case a MCM is any algorithm where a numeric solution is estimated based on the results of repeated sampling. The experiments are defined and implemented according to the following methodology;

- Define a suitable input domain.
- Generate random inputs from a suitable probability density function (PDF) over the input domain.
- Perform necessary calculation on the inputs.
- Aggregate results

The most common implementations of this method are to define the MCM explicitly as the solution to an integral, (Monte Carlo as Quadrature), or to design a simulation-based approach to estimate the solution, (Monte Carlo as Simulation).

Given a continuous function with a single variable $z(x)$, dependent on a random variable x, the mean or expected value of $z(x)$ is given by:

$$\langle z \rangle = \int_{[0,1)^s} z(x) f(x) \mathrm{d}x \tag{3.1}$$

where $x \in [0,1)^s$ and $f(x)\mathrm{d}x$ is the probability that x has a value within dx about x. Monte Carlo methods are used when integrals of this type may not be evaluated analytically and instead an estimate for $\langle z \rangle$ is required. Applying a quadrature scheme an estimate for the value of $\langle z \rangle$ is found by summing a set of weighted evaluations of the integrand:

$$\langle z \rangle \approx \sum_{i=1}^{N} w_i z(x_i) f(x_i) \tag{3.2}$$

where w_i are the weights and x_i are the nodes or abscissas of the quadrature scheme. Applying a basic Monte Carlo approach a set of N abscissas are generated according

to the PDF $f(x)$ which are used to determine the value of the sample mean:

$$\bar{z} \equiv \frac{1}{N} \sum_{i=1}^{N} z(x_i) \tag{3.3}$$

which, (according to the central limit theorem), approximates the solution to $\langle z \rangle$ given a suitable sample size N:

$$\lim_{N \to \infty} \bar{z} = \langle z \rangle \tag{3.4}$$

In the case of a basic Monte Carlo approach the weight values are set such that $w_i = \frac{1}{Nf(x_i)}$ and the PDF is not included in $\bar{z}$ [46].

3.2.3 Sampling Methods

The concept of convergence is an important metric used to determine performance and accuracy of a MCM estimator. As such reducing the effort required to obtain a strong result is a high priority when developing MCMs. Given an integral as in equation 3.1 and an MCM estimator as in equation 3.3:

$$\langle z \rangle = \int_{[0,1)^s} z(x) f(x) \mathrm{d}x \approx \bar{z} = \frac{1}{N} \sum_{i=1}^{N} z(x_i) \tag{3.5}$$

the standard deviation of the sample mean $\bar{z}$ is given by:

$$s(\bar{z}) = \frac{1}{\sqrt{N-1}} \sqrt{\overline{z^2} - \bar{z}^2} \tag{3.6}$$

As the values $\overline{z^2}$ and $\bar{z}^2$ must always be positive, variance reduction methods are typically aimed at minimizing the value of $\overline{z^2} - \bar{z}^2$ [46]. Some of the most widely used methods are modified sampling methods. Using these methods the sampling methodology is biased by modifying the sampling PDF. These types of methods include *Importance Sampling*, *Stratified Sampling* and *Correlated Sampling with Antithetic*

Variates, each of which is discussed in brief in this section.

Importance Sampling

Given the standard MCM estimator given in equation 3.5 consider a system in which the values of $z(x)$ of interest to the result are distributed over a small sub-region of the overall volume. In this scenario, the PDF $f(x)$ sampling uniformly over the region will produce many results that are not of interest and in fact can be discarded. Given the modern implementation of MCMs as computer simulations this corresponds to a potentially waste of computational resources, alternatively, the efficiency of the system can be greatly improved by sampling only over a sub-region of interest. This is achieved by defining an arbitrary PDF $f^*(x)$ and adjusting the expected value:

$$\langle z \rangle = \int_{[0,1)^s} \frac{z(x)f(x)}{f^*(x)} f^*(x)\mathrm{d}x \tag{3.7}$$

$$= \int_{[0,1)^s} z^*(x)f^*(x)\mathrm{d}x \tag{3.8}$$

$$= \langle z^* \rangle \tag{3.9}$$

where $z^*(x) = z(x)W(x)$ and $W(x) = \frac{f(x)}{f^*(x)}$ is a weight function used to remove bias introduced by sampling from $f^*(x)$. The sample mean of the estimator is now calculated as follows:

$$\langle z \rangle = \langle z^* \rangle \tag{3.10}$$

$$\approx \overline{z^*} = \frac{1}{N}\sum_{i=1}^{N} z(x_i)W(x_i) \tag{3.11}$$

where x_i are sampled from $f^*(x)$. The variance of $\overline{z^*}$ is given by:

$$\sigma^2(\overline{z^*}) = \langle z^{*2} \rangle - \langle z^* \rangle^2 \tag{3.12}$$

$$= \langle z^{*2} \rangle - \langle z \rangle^2 \tag{3.13}$$

Note that $\langle z^* \rangle^2 = \langle z \rangle^2$, but $\langle z^{*2} \rangle \neq \langle z^2 \rangle$:

$$\langle z^{*2} \rangle = \int_{[0,1)^s} z^{*2}(x) f^*(x) \mathrm{d}x \tag{3.14}$$

$$= \int_{[0,1)^s} z^2(x) W^2(x) f^*(x) \mathrm{d}x \tag{3.15}$$

$$= \int_{[0,1)^s} z^2(x) \frac{f^2(x)}{f^*(x)} \mathrm{d}x \tag{3.16}$$

$$= \int_{[0,1)^s} z^2(x) \frac{f(x)}{f^*(x)} f(x) \mathrm{d}x \tag{3.17}$$

$$= \int_{[0,1)^s} z^2(x) W(x) f(x) \mathrm{d}x \neq \int_{[0,1)^s} z^2(x) f(x) \mathrm{d}x \tag{3.18}$$

Given the inequality in 3.18 the variance of the expected result can be reduced by implementing a weight function $W(x) < 1$ for regions of $z(x)$ that affect the the expected value [46].

Stratified Sampling

Stratified sampling is an implementation of Systematic Sampling. Using these methods the integral region is divided into a set of sub-regions, and the variance of the final estimator may be reduced by determining how many samples to take from each sub-region. In the case of Stratified Sampling, the number of samples taken from each region is proportional to the variance of the integral over that sub-region - more samples will be taken from sub-regions with higher variance. Given the integral z over a volume $V = [0,1)^s$, a stratified sampling method defines a set of M sub-regions over V such that:

$$\langle z \rangle = \sum_{m=1}^{M} \int_{V_M} z(x) f(x) \mathrm{d}x \tag{3.19}$$

The probability of a sample x_i from PDF $f(x)$ landing in a sub-region V_M is defined as $p_m = \int_{V_M} f(x) \mathrm{d}x$ where $\sum_{m=1}^{M} p_m = 1$, with this definition, a set of PDFs for each

sub-region may be defined:

$$f_m(x) = \begin{cases} \frac{f(x)}{p_m} & \text{if } x \in V_m \\ 0 & \text{otherwise} \end{cases} \tag{3.20}$$

Using these PDFs the expected value is now defined as:

$$\langle z \rangle = \sum_{m=1}^{M} p_m \int_{V_M} z(x) f_m(x) \mathrm{d}x \tag{3.21}$$

and the sample mean of the estimator as:

$$\overline{z} = \sum_{m=1}^{M} p_m \overline{z_m} \tag{3.22}$$

where:

$$\overline{z_m} = \frac{1}{N_m} \sum_{i=1}^{M} z(x_i) \tag{3.23}$$

and the number of samples taken from each region N_m is proportional to $\sigma^2(z)$ so long as $\sum_{m=1}^{M} N_m = N$:

$$N_m = \frac{p_m \sigma_m(z)}{\sum_{m=1}^{M} p_m \sigma_m(z)} N \tag{3.24}$$

Compared with simplified random sampling, stratified sampling is known to reduce the variance of the estimated value in most cases, however if the variance of z is constant across V then there will be no improvement. Furthermore the method requires that the variance of z in each sub-region be known in advance in order to determine the number of samples to draw from each. Alternatively, if the variance is unknown a preliminary series of trials may be performed in order to estimate the sample variance of z, and this value may be used to determine N_m [46].

Correlated Sampling with Antithetic Variates

Correlated sampling methods are used when performing a Monte Carlo simulation comparing two almost equal scenarios. Using correlated sampling, instead of performing two independent simulations, each simulation is performed using the same random number sequence and the difference is calculated to determine the final result. By using the same random numbers the two simulations are highly correlated reducing the variance in the final result. Given a system $\langle \Delta z \rangle = \langle z_1 \rangle - \langle z_2 \rangle$ where:

$$\langle z_1 \rangle = \int_V z_1(x) f_1(x) \mathrm{d}x \tag{3.25}$$

$$\langle z_2 \rangle = \int_V z_2(x) f_2(y) \mathrm{d}x \tag{3.26}$$

the estimator of the result is given by:

$$\overline{\Delta z} = \overline{z_1} - \overline{z_2} \tag{3.27}$$

$$= \frac{1}{N} \sum_{i=1}^{N} z_1(x_i) - \frac{1}{N} \sum_{i=1}^{N} z_2(y_i) \tag{3.28}$$

the variance of the result is given by:

$$\sigma^2(\langle \Delta z \rangle) = \sigma^2(\langle z_1 \rangle) + \sigma^2(\langle z_2 \rangle) - 2\,\mathrm{cov}(\overline{z_1}, \overline{z_2}) \tag{3.29}$$

If $\overline{z_1}$ and $\overline{z_2}$ are calculated independently then $\mathrm{cov}(\overline{z_1}, \overline{z_2}) = 0$, but if the random variables x and y are positively correlated then $\mathrm{cov}(\overline{z_1}, \overline{z_2}) > 0$ reducing the variance in the final estimator. Antithetic variates are a special case of correlated sampling where the simulations used for the calculation of the sample mean are performed with two simulation paths, the first using the set of random numbers $X_1 = x_1, ..., x_n$ and the second using $X_2 = -x_1, ..., -x_n$, thus creating the antithetic path. Performing a simulation of $\langle z \rangle$ using these two paths to implement the two estimators $\overline{z_1}$ and $\overline{z_2}$

allows for the final estimator to be found using

$$\bar{z} = \frac{\overline{z_1} + \overline{z_2}}{2} \tag{3.30}$$

The estimators $\overline{z_1}$ and $\overline{z_2}$ are calculated using the standard method for MCMs:

$$\overline{z_1(x)} = \frac{1}{N}\sum_{i=1}^{N} z(x_i) \tag{3.31}$$

$$\overline{z_2(x)} = \frac{1}{N}\sum_{i=1}^{N} z(1-x_i) \tag{3.32}$$

Substituting these two equations into 3.30 gives the following:

$$\bar{z} = \frac{1}{2N}\sum_{i=1}^{N} \left[z(x_i) + z(1-x_i)\right] \tag{3.33}$$

The variance in the result is given by:

$$\sigma^2(z) = \frac{1}{4}\left[\sigma^2(z_1) + \sigma^2(z_2) + 2\,\mathrm{cov}(z_1, z_2)\right] \tag{3.34}$$

As is the case for standard correlated sampling methods, a strong negative correlation between the estimators $\overline{z_1}$ and $\overline{z_2}$ will reduce the variance in the result estimator [46].

3.3 Quasi Monte Carlo Methods

3.3.1 Definition

A qMCM [73, 152, 169] is a special class of MCM that uses a low discrepancy sequence (LDS) to generate the random number sequences used for the simulation as opposed to a pseudo-random number sequence. qMCMs are preferred over standard MCMs in certain situations due to the fast convergence property of the method [33, 134]. This fast convergence is achieved using a LDS, defined as a sequence with significantly lower discrepancy than that of a typical random number set [107]. The benefit of lower discrepancy is the higher degree of distribution in

finite random number sets with small sizes, pseudo-random numbers will achieve uniform distribution as the set size, n, approaches infinity. This, in conjunction with the central limit theorem, allows for a suitable estimator, $\bar{z}$, of an expected value, $\langle z \rangle$, to be found as part of a Monte Carlo Simulation, given a suitable number of trials, N:

$$\bar{z} = \frac{1}{N} \sum_{i=1}^{N} z(x_i) \tag{3.35}$$

$$\lim_{N \to \infty} \bar{z} = \langle z \rangle \tag{3.36}$$

where the inputs x_i are generated by a suitable random number generator. When using pseudo-random number generators equi-distribution of the random number sequence is achieved as $N \to \infty$. Alternatively a LDS can be designed to achieve equi-distribution for finite values of N. This reduces the variance in the estimator and as such improves the convergence rate of the estimated result to the true result. These types of sequences are used when the aim is not to create a truly random sequence of numbers, but to create an approximately even distribution of values throughout the input space of the simulation. In certain circumstances and with careful choice of sequence type the faster convergence properties of a qMCM are useful for simulations where the ability to perform the simulation using smaller sample sizes is a priority [33, 84].

3.3.2 Measuring Discrepancy

The measurement of order in a random number sequence is referred to as the discrepancy of the number sequence, this measurement is closely tied to the distribution, or more accurately the equi-distribution, of a number sequence [2, 42]. A bounded sequence, is this particular case a bounded random number sequence, can be said to be equi-distributed over a particular interval if the number of values within that interval is proportional to the length of the interval. More formally, a bounded sequence $\{x_1, x_2, ..., x_N\}$ is equi-distributed over the interval $[a, b]$ if for any subinterval

$[c,d]$ the following applies [2]:

$$\lim_{N\to\infty} \frac{|\{x_1, x_2, ..., x_N\} \cap [c,d]|}{N} = \frac{d-c}{b-a} \tag{3.37}$$

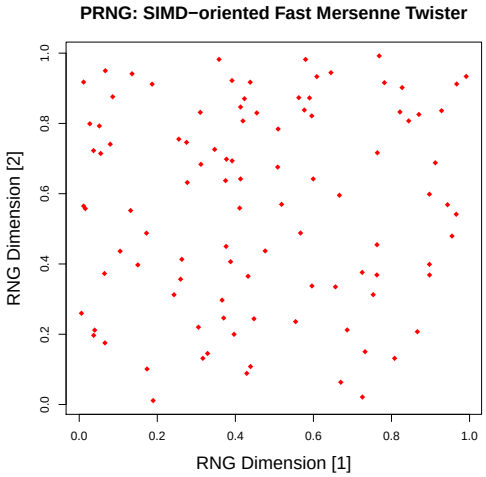
The above equation can be modified to calculate the discrepancy of a sequence:

$$D([c,d];N) = \limsup_{a\leq c\leq d\leq b} \left| \frac{|\{x_1, x_2, ..., x_N\} \cap [c,d]|}{N} - \frac{d-c}{b-a} \right| \tag{3.38}$$

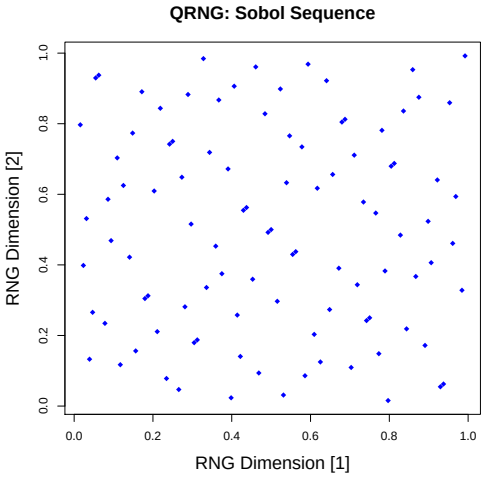
From the above equation it can be seen that it relates to equation 3.37 in that a sequence can be defined as equi-distributed if $D \to 0$ as $N \to \infty$, it can then be stated that to obtain a low discrepancy sequence across the range $[a,b]$ then the value of D for this sequence should be minimized. In order to obtain a more general measurement of discrepancy the *star discrepancy* of a sequence can be calculated:

$$D^*(N) = \max_{[c,d]} |D([c,d];N)| \tag{3.39}$$

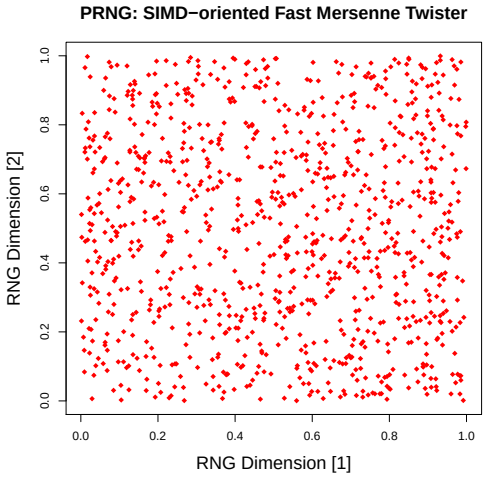
The star discrepancy determines the region of maximum discrepancy, or minimum equi-distribution, across the entire sequence, as opposed to D(N) which measures the discrepancy of a specific region. These formulas can essentially be seen as breaking the sequence up into smaller and smaller subregions and measuring the ratio of the number of points in the subregion to the relative range of the subregion.



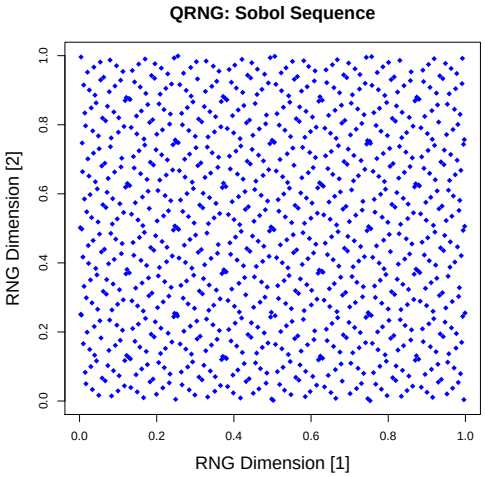
(a) PRNG - Two-dimension SFMT at $N = 100$ iterations



(b) QRNG - Two-dimension Sobol Sequence at $N = 100$ iterations



(c) PRNG - Two-dimension SFMT at $N = 1000$ iterations



(d) QRNG - Two-dimension Sobol Sequence at $N = 1000$ iterations

Figure 3.1: Comparison of Pseudo-Random and Quasi-Random Number Sequence in 2 Dimensions

3.3.3 Pseudo-Random v. Quasi-Random

Semantically a LDS is differentiated from a pseudo-random sequence using the term quasi-random sequence, leading to the distinction between a MCM and a qMCM. Pseudo-random sequences are by definition not truly random sequences, as it is difficult to recreate true randomness within a digital system. Some elements of determinism will normally be required to produce a sequence of real numbers. For this reason random number generators are designed to produce a sequence of values based on one or more *seed values* that determine the sequence, i.e. the sequence generated will always be the same for a particular seed value but will appear random. In order to increase the random properties of a sequence the seed value can be based on values like time, date or the value of an arbitrarily selected register. A pseudo-random sequence, due to its random nature, does not maintain an even distribution of random values, the distribution of values simply becomes more equi-distributed as more values are produced. Alternatively the goal of a quasi-random generator is to ensure that equi-distribution is achieved from the smallest n-value possible, this has the effect of making the sequence much less random, but ensuring an even spread of values at all times [3, 42]. This is achieved by generating new values that are as far away from the other values in the sequence as possible, thus avoiding clustering of values within the sequence and maintaining equi-distribution of the sequence. The difference between pseudo-random and quasi-random sequences is visible when plotted using two-dimensional sequences, as shown in figure 3.1. These plots have been made using random number generators available in R, the pseudo-random sequence is generated using a single instruction multiple data (SIMD)-orientated fast Mersenne twister [121, 153], and the quasi-random sequence is generated using a Sobol sequence [13, 159]. Both random number generators are available as part of the R package `randtoolbox` [47]. In these figures are set of two-dimensional random number sequences are plotted for $N = 100$ iterations in figures 3.1(a) and 3.1(b), then $N = 1000$ iterations in figures 3.1(c) and 3.1(d). In the pseudo-random sequences shown on the left of the figure, areas with proportionally less points than others are

visible, indicating a low level of equi-distribution and high discrepancy within the sequence. Comparing the $N = 100$ and $N = 1000$ cases it can be seen that the level of equi-distribution increases with N. The quasi-random sequences shown on the right side of the figure show a high level of order and equal distribution of points for both $N = 100$ and $N = 1000$ iterations.

3.3.4 Effect on Rate of Convergence

The overall benefit of qMCM is the fast convergence rate of the simulation. The approximation error of a Monte Carlo estimation is given by:

$$\epsilon = |\langle z \rangle - \bar{z}| \tag{3.40}$$

$$= \left| \int_{[0,1)^s} z(x) f(x) \mathrm{d}x - \frac{1}{N} \sum_{i=1}^{N} z(x_i) \right| \tag{3.41}$$

In the case of standard MCMs the upper bound of this error is known to be proportional to $\frac{1}{\sqrt{N}}$, whereas in the case of qMCMs the error is proportional to the discrepancy of the input sequence $X = \{x_1, ..., x_i\}$ and is bounded by:

$$|\epsilon| \leq V(z) D_N \tag{3.42}$$

where $V(z)$ is the Hardy-Krause variation of the function z and D_N is the discrepancy of the random number set. This may be used to show that the approximation error in a quasi-Monte Carlo simulation is proportional to $\frac{\log(N)^s}{N}$, where s is the number of dimensions in the random number space. Although it is only possible to determine the upper bound of the approximation error, (i.e. the worst case convergence rate of the method), in practice qMCM will converge significantly faster than standard MCMs and in fact can achieve a convergence rate near to $\frac{1}{N}$ [7].

3.3.5 Randomized Quasi-Monte Carlo Methods

There are several limitations that must be considered in order to justify the use of qMCMs. In addition to the fact that only the upper bound on the approximation error is known, for many functions $V(z) = \infty$ and both $V(z)$ and D_N can be difficult to compute. In addition, to ensure that the convergence rate is significantly lower, i.e:

$$\mathrm{O}\left(\frac{\log(N)^s}{N}\right) \ll \mathrm{O}\left(\frac{1}{\sqrt{N}}\right) \tag{3.43}$$

the total number of dimensions must be small and the total number of samples required increases significantly with s. These limitations on qMCMs can be mitigated using an extension known as the randomized quasi-Monte Carlo method (RqMCM). The qMCM can be seen as a deterministic method rather than random due to the use of LDSs, leading to the inability to determine the variance and making the upper bound on the approximation error difficult to estimate. Randomizing the method allows for the variance and error to be calculated in order to assess the effectiveness of qMCMs. However, several conditions must hold in order to guarantee that the estimated result is an unbiased estimation of the true result, and that the desirable properties of the original qMCM are maintained. Given the following definitions for an expected value and the sample mean of its estimator:

$$\langle z \rangle = \int_{[0,1)^s} z(x) f(x) \mathrm{d}x \tag{3.44}$$

$$\overline{z} = \frac{1}{N} \sum_{i=1}^{N} z(x_i) \tag{3.45}$$

$$\lim_{N \to \infty} \overline{z} = \langle z \rangle \tag{3.46}$$

the LDS $X_N = x_1, ..., x_N \in [0,1)^s$ may be randomized in order to form the sequence $\widetilde{X_N}$. This new sequence will be uniformly distributed over $[0,1)^s$, while still maintaining the equi-distribution, (low discrepancy) of the sequence X_N. This guarantees

that $\bar{z}$ is an unbiased estimator of $\langle z \rangle$ and allows the variance may now be estimated and compared with standard Monte Carlo methods [111, 112]. The simplest form of randomization that is applicable to LDSs [129, 164] is a randomly shifted estimator [108]. This type of sequence is formed by taking Y_N a s dimension random vector uniformly distributed over $[0,1)^s$ and adding it to the original LDS, X_N, and applying a modulo one operation to form the result vector. Thus the expected value of the integral $\langle z \rangle$ is calculated as follows:

$$\bar{z} = \frac{1}{N} \sum_{i=1}^{N} z((x_i + y_i) \mod 1) \tag{3.47}$$

Other types of randomization include $\mathbb{B}$-ary Digital Shifts, Scrambling and Random Linear Scrambling [108].

3.4 Monte Carlo Arithmetic

MCA [142] is an application of the MCM to error analysis in FP arithmetic that allows for the sensitivity to rounding error of a FP operation or series of operations to be measured. MCA tracks rounding errors at run-time by applying randomization to input and output operands forcing the results of FP operations to behave like random variables. This turns an execution into trials of a Monte Carlo simulation allowing statistics on the effects of rounding error to be obtained over a number of executions. Statistical measurements are then used to analyse the results, sensitivity to rounding error is suspected if a high level of variance is observed between trials. As an example, consider again the polynomial presented in Section 2.6.4:

$$x^2 + 444x + 1 = 0 \tag{3.48}$$

solved with the quadratic formula to determine the roots r_1 and r_2. To perform a MCA simulation on the formula a set of N executions is performed, replacing the original Institute of Electrical and Electronic Engineers (IEEE)754 FP operations with

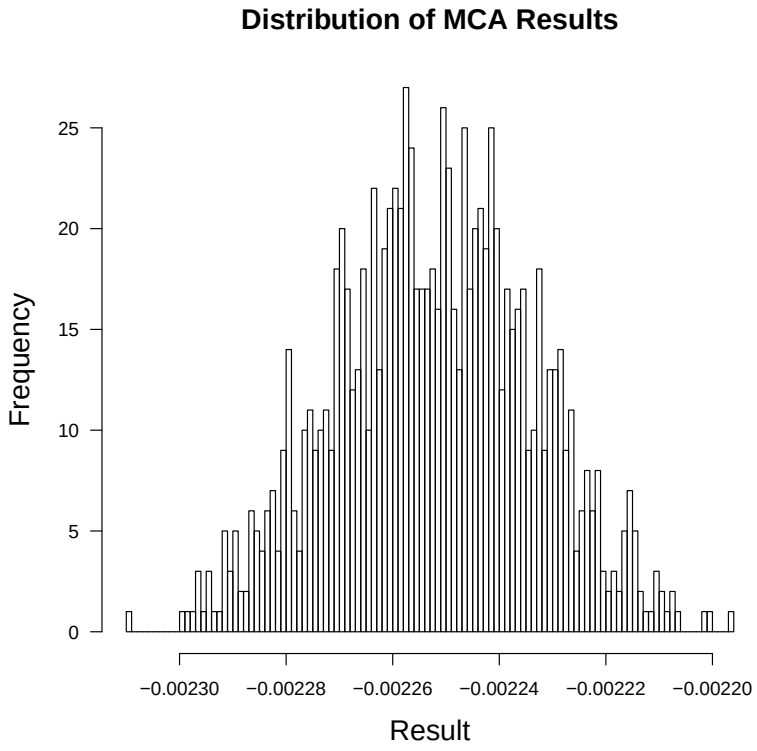


Figure 3.2: Distribution of results for non-stable system

MCA operations, randomizing the input and output operands at each step in the execution. Due to the instability of the result a large number of the digits in the result are randomized, creating a large variance in the results of the simulation, as shown in Figure 3.2. By contrast, simulating the solution to a more stable result, such as the solution to the following polynomial:

$$x^2 - 1 = 0 \tag{3.49}$$

produces a set of results with significantly lower variance. The sample mean and variance for both solutions is detailed in Table 3.1 and compared with the results obtained using standard IEEE754 FP operations.

$x^2 - 1$		
	r_1	r_2
Known Result	1	-1
Sample Mean	1	-1
Sample Variance	1.7736×10^{-07}	1.7856×10^{-07}
$x^2 + 444x + 1$		
Known Result	-0.00225226368	-443.997747736
Sample Mean	-0.002252174	-443.9977
Sample Variance	1.8620×10^{-05}	6.6117×10^{-05}

Table 3.1: Monte Carlo Arithmetic - Example results for systems sensitive and in-sensitive to rounding error. Example results obtained using double precision IEEE-754 operators and virtual precision $t = 24$.

3.4.1 Modelling Inexact Values

The finite precision requirements of computer arithmetic systems results in the inevitability of inexact values within the results of a computation. As these values must be rounded to the nearest exact value, this leads to the inevitability of rounding error. Although some standards, such as IEEE754, have an exception flag to indicate inexactness in a value, this flag is ignored in most cases. Floating point standards do not allow for information on inexactness to be tracked throughout a computation, and as such rounding an inexact value results in the loss of information on the inexact nature of that value. This essentially forces floating point operators to treat all operands as exact values. Using MCA inexactness in floating point variables is simulated using a random variable, allowing the effects of rounding error and inexactness to propagate through a floating point computation. By controlling the way in which random perturbations are applied to operands the results of arithmetic operations are randomized in a deterministic fashion and repeated evaluations will produce differing results. This turns each execution into a trial of a Monte Carlo simulation and the results may now be evaluated statistically. Using MCA the inexactness of a FP operand is modelled using the *inexact* function [142, p. 32]. If x is a non-zero FP value of the form given in Equation 2.44 the inexact function is

defined as follows:

$$\text{inexact}(x, t, \xi) = x + \beta^{e_x - t}\xi \tag{3.50}$$

$$= (-1)^{s_x}(m_x + \beta^{-t}\xi)\beta^{e_x} \tag{3.51}$$

where $x \in \mathbb{R}$, t is a positive integer representing the desired precision, ξ is a uniformly distributed random variable in the range $[-\frac{1}{2}, \frac{1}{2})$, ($\xi \in U[-\frac{1}{2}, \frac{1}{2})$) and m_x, e_x are the mantissa and exponent of x. It is assumed that $0 < t \leq p$. An operation $\circ \in \{+, -, \times, \div\}$ is implemented as [142, p. 38]:

$$x \circ y = \text{round}(\text{inexact}(\text{inexact}(x) \circ \text{inexact}(y))) \tag{3.52}$$

Using the methods shown in the previous equation a random perturbation is applied to both the incoming operands and the result of the operation. Randomization of inputs is referred to as *precision bounding*, while randomization of the output is referred to as *random rounding*, these two techniques are discussed in the next two sections.

3.4.2 Precision Bounding

Precision bounding of an operation during MCA is used for the detection of catastrophic cancellation [142, 144, 181], which occurs when subtracting two similar operands, (i.e when the result of the operation has a smaller exponent value than either operand). This type of subtraction will result in a high number of leading zeroes in the mantissa before normalization is performed, and after normalization a high number of zeroes will be inserted during the right shift operation. As these are assumed zeroes they cannot be seen as significant digits causing a loss of accuracy in the result. The application of precision bounding will insert random digits behind the significant digits of the result during the operation, applying randomization during the normalization stage. This has the effect of applying one random digit

for every significant digit lost during normalization, if a high number of significant digits are lost then the amount of randomization applied will be larger, this allows for catastrophic cancellation to be detected by measuring the relative standard deviation of the results of a Monte Carlo Simulation. The value of t is used to determine the level of random perturbations applied to the operation, and is referred to as the *virtual precision* of the operation. In the case of precision bounding the value of t will determine the level of significant digits to which a randomized operand agrees with the original value, i.e. a precision bounded operand:

$$\overline{x} = x \pm \beta^{1-t}\xi \tag{3.53}$$

will be equal to the original value x to t digits. In the case of IEEE754 single precision arithmetic $p = 24$ and subsequently $0 < t \leq 24$. If t is set to 24 and an operand is precision bounded, then that operand's new value will be equal to the original value up to 24 digits, if t is set to 12 then the values will only be equal to 12 digits, effectively multiplying the random perturbation applied by 2^{12}. Precision bounding of an operand can be applied in one of several ways. The first method is to extend the precision of the incoming operands by doubling the length of the mantissa to $2p$, then applying random digits behind the original mantissa using fixed point arithmetic, effectively performing the operation shown in equation 3.53. The second method is to modify the shift module used during the operation to place random digits onto the mantissa as the shift operation is performed. Both of these methods require modifications to the internal structure of the FPU.

3.4.3 Random Rounding

Random rounding is characterized as the precision bounding of a floating point output, used to eliminate round-off error in an operation by modelling forward error and ensuring zero rounding bias over a set of operations [142, 144]. The operation is

performed in terms of the inexact function as follows:

$$x \circ y = \text{round}(\text{inexact}(x \circ y)) \tag{3.54}$$

$$= \text{round}(x \circ y \pm \xi) \tag{3.55}$$

This equation can be simplified by substituting the operation values $x \circ y$ for the un-rounded and un-normalized result of this operation and determining a value for the rounding error ξ for each operation, as shown below:

$$x \pm y = \text{round}(x \pm y + \xi\beta^{e_x - t}) \tag{3.56}$$

$$x * y = \text{round}(x * y + \xi\beta^{e_x + e_y - t}) \tag{3.57}$$

$$\frac{x}{y} = \text{round}(\frac{x}{y} + \xi\beta^{e_x - e_y - t}) \tag{3.58}$$

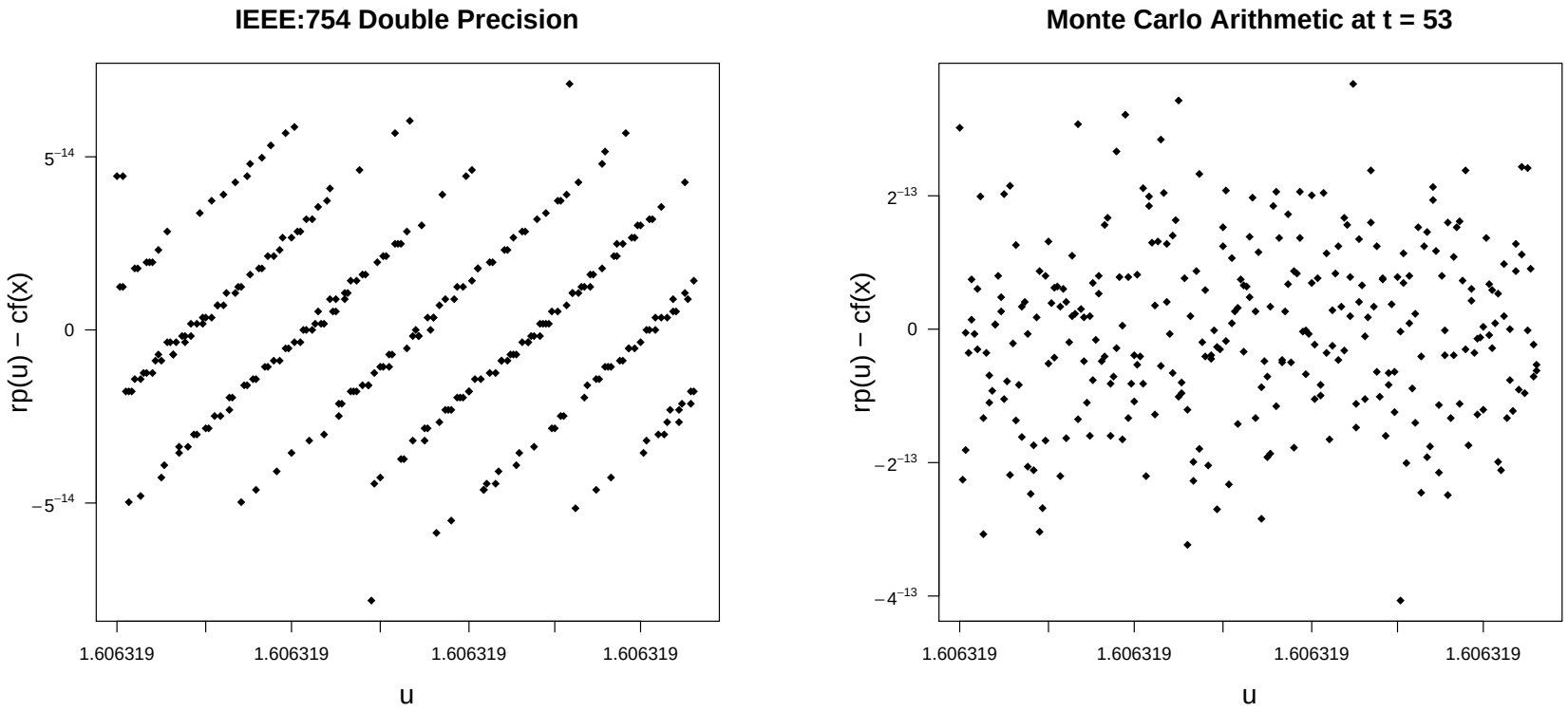
where ξ is a uniformly distributed random value $\xi \in U[-\frac{1}{2}, \frac{1}{2})$. Performing random rounding, or precision bounding of results, will model errors within the operator known as forward error, as opposed to precision bounding of inputs which models error in the operands or the representation (precision) of floating point values, known as backward error. The application of random rounding will force the rounding of an operation to have zero round-off bias over a set of operations, as the round-off errors become random and un-correlated. By forcing round-off error to be randomized the expected error from round-off can be eliminated by averaging the results of n trials. This effect also has the benefit of providing evidence to the benefit of MCA. Studies in the past including [82, 97] have stated that statistical analyses of round-off error in computer arithmetic are unfounded when they assume rounding errors are random. When considering functions that are sensitive to input perturbation using standard IEEE754 arithmetic, rounding errors are often non-random and correlated. An example presented in both [97, 142] demonstrates this effect using the following example where two forms of the same function, (rational polynomial vs. continuous

fraction) are compared to determine their sensitivity to small input perturbations:

$$cf(x) = 4 - \frac{3 \cdot (x-2) \cdot ((x-5)^2 + 4)}{x + (x-2)^2 \cdot ((x-5)^2 + 3)} \tag{3.59}$$

$$rp(x) = \frac{622 - x \cdot (751 - x \cdot (324 - x \cdot (59 - 4x)))}{112 - x \cdot (151 - x \cdot (72 - x \cdot (14 - x)))} \tag{3.60}$$

The value of $rp(u) - cf(x)$ is calculated for $x = 1.60631924$, $u = x, x + \epsilon, ..., x + 300\epsilon$ and $\epsilon = 2^{-53}$. Results obtained using standard IEEE754 double precision operators are shown in Figure 3.3(a) where it can be seen that the rounding errors in the function do not behave like random variables. When the results are re-run using MCA with a virtual precision of $t = 24$, the rounding errors are now randomized as can be seen in Figure 3.3(b).



(a) Results using IEEE754 Double Precision

(b) Results using MCA with $t = 53$

Figure 3.3: Distribution of results for operations sensitive to input perturbation - MCA v. IEEE double precision [97, 142].

3.4.4 Virtual Precision t

An important concept within MCA is the *virtual precision* or t value of an operation. This value determines the level of random perturbation applied to an input or result, and may be used to determine the minimum precision required to perform an operation to a specified level of accuracy [142, 144]. The value of t as used in the inexact function:

$$\text{inexact}(x) = x \pm \beta^{e_x+(1-t)}\xi \tag{3.61}$$

will determine the size of the random value relative to the original operands by determining the level to which the random value ξ is shifted to the right. When $t = p$, the ξ value is shifted a total of p places, i.e. it will be appended to the end of the mantissa m_x. This will also result in p significant digits in the value x, i.e $inexact(x) = x$ to p digits. By varying the value t the number of significant digits in the operand will also be varied and subsequently the accuracy of the operation is controlled, this feature results in variable precision MCA and can be used to determine the minimum precision p required to perform an operation accurately. This type of testing is performed by obtaining a set of results for increasing values of t, starting with $t = 1$ and increasing until $t = p$. At each t value n samples are obtained and analysis of the results is performed to determine the sensitivity to rounding error and the number of stable significant figures in the results. Using variable precision MCA an algorithm can be tested to determine a required precision that is tailored to not only the specific algorithm, but using field programmable gate array (FPGA) and hardware acceleration techniques tailored to the specific hardware configuration. This is of particular use in the field of application specific integrated circuit (ASIC) design as high efficiency is required due to limitations on area and performance, the ability to reduce floating point format sizes by determining minimum required precision allows for the most efficient format to be determine for the specific design being implemented.

3.5 Summary

MCA is an implementation of the MCM applied as an extension to FP arithmetic. The MCM is itself a development of Statistical Sampling methods made possible with modern computer systems. Using a MCM the expected value, $\langle z \rangle$, of a function $z(x)$ may be approximated by performing a series of trials with a randomized input and determining the sample mean $\overline{z}$:

$$\langle z \rangle = \int_{[0,1]^s} z(x) f(x) \mathrm{d}x \tag{3.62}$$

$$\overline{z} = \frac{1}{N} \sum_{i=1}^{N} z(x_i) \tag{3.63}$$

$$\lim_{N \to \infty} \overline{z} = \langle z \rangle \tag{3.64}$$

The application of the MCM to the issue of FP error analysis is intended to model inexactness in FP values and determine the effect of mixed forward/backward error on the results of FP operations. As such the function to be approximated by simulation is based on the error model presented in Section 2.6.1:

$$\hat{z} = f(\hat{x}, \hat{y}) \tag{3.65}$$

$$z(1 + \delta_z) = (x(1 + \delta_x) \circ y(1 + \delta_y)) \tag{3.66}$$

$$\tag{3.67}$$

where $\hat{x}$, $\hat{y}$ and $\hat{z}$ are the rounded approximations of the exact values x, y and z and $\circ \in \{+, -, \times, \div\}$ a FP operation. The approximation (forward) errors in $\hat{x}$ and $\hat{y}$ are represented by the values δ_x and δ_y, and the rounding (backward) error in the result is represented by δ_z. Using variable precision MCA this model is implemented for the purposes of Monte Carlo as Simulation using the inexact function:

$$\text{inexact}(x, t, \xi) = x + \beta^{e_x - t} \xi \tag{3.68}$$

The sample mean $\bar{z}$ is determined after a set of N trials where an individual result of a FP operation is determined as follows:

$$z_i = \text{round}(\text{inexact}(\text{inexact}(x) \circ \text{inexact}(y))) \tag{3.69}$$

The application of the inexact function to the inputs x and y is referred to as precision bounding and models the forward error in the values $\hat{x}$ and $\hat{y}$ respectively, while application of the inexact function to the result is referred to as random rounding and models the backward error in the operation due to rounding. By re-implementing the basic FP operations the effects of inexactness and rounding error on mathematical software may now be modelled. Repeated executions will in turn generate a set of Monte Carlo simulations that may now be analysed using standard statistical methods, in particular, measuring the variance in a set of results may determine the sensitivity to rounding error and the number of stable significant figures in the results.

One of the primary drawbacks of the MCM is the requirement for repeated executions, in the case of MCA results from Parker and experimental results presented in the following chapters demonstrates that sample sizes in the order of 100 executions are typically required. When applied to FP arithmetic and associated mathematical software, an area where speed is typically a key performance metric, the reduction in performance is a significant factor. At present a number of variance reduction schemes designed to improve the rate of convergence of Monte Carlo results have been developed and applied to standard implementations of the MCM however these have not yet been applied to MCA.

Chapter 4

MCALIB - A Tool for Automated Rounding Error Analysis

4.1 Introduction

Despite the advantages offered by Monte Carlo arithmetic (MCA) and similar techniques, tools for rounding error analysis are not in common usage. One of the major barriers is that source code needs to be modified so that custom libraries are called to execute the arithmetic operations. In this work, the use of source to source compilation, supported by mixed precision libraries, is advocated. The approach allows for the implementation of a general purpose floating point (FP) analysis tool that can be applied to arbitrary programs without significant changes to the source code, a technique that we refer to as Monte Carlo programming (MCP). The implementation provides opportunities for wider adoption of runtime error analysis, and allows developers to test both the accuracy of algorithms and the suitability of different FP formats for a particular implementation. Although our tool is designed to be used with MCA, the same approach could be used in conjunction with other rounding analysis techniques. MCP can be used for the simplified implementation of several data analysis schemes, such as sensitivity analysis to measure the effect of uncertainty in input data or arithmetic operations. The ef-

fect of missing data, dirty data and inexact data can also be measured. An open source implementation of Monte Carlo arithmetic library (MCALIB), including C intermediate language (CIL) libraries and documentation, is available via github from https://github.com/mfrechtling/mcalib.git. The remainder of this chapter is organized as follows. The implementation of the library is detailed in Section 4.2. Methods for interpreting the results of MCA analysis are provided in Section 4.3. Section 4.4 describes test cases and methods. Results are presented in Section 4.5, and finally, conclusions are drawn in Section 4.6.

4.2 MCALIB Implementation

4.2.1 Source-to-Source Compilation

Source-to-source compilation provides an effective tool for automated code transformations [54], and when paired with error analysis techniques allows for the implementation of automated software (SW) verification [88, 133]. The CIL [132] is a high level language representation, including a set of tools for analysis and source-to-source compilation of C programs. The CIL compiler **cilly** is implemented as a Perl script that performs translations to C code as defined in a set of OCaml modules provided as part of the CIL library. For the purposes of MCALIB, CIL has been used for transforming C FP operations into calls to the MCALIB library. This has been done by first lowering the source code to a single statement assignment form, then converting FP operations to use MCALIB library functions. As an example the following single precision multiplication operation:

```
a = b * c;
```

would be redefined to the following function call:

```
a = _floatmul(b, c);
```

where `float _floatmul(float a, float b)` is the MCALIB function for handling single precision MCA multiplication. This process will result in all supported

FP operations being replaced with function calls to the MCALIB library. It is important to note that although operations are done in a higher precision, the storage requirements of the FP variables remain unchanged. This avoids portability issues associated with pointers and dynamic memory allocation.

4.2.2 Library Implementation using MPFR

MCA has been implemented within MCALIB as a set of library functions for arithmetic and comparison operations. One of the main difficulties with implementing MCA is the need to extend the precision of the FP format being tested in order to simulate infinite precision. The precision level must include p machine bits and t virtual bits, a total precision requirement of $W = p + t$, where W is the working precision of the MCA operation. The MCALIB library also implements variable precision MCA, allowing the virtual precision to vary between $0 \leq t \leq p$ at runtime. To achieve this functionality the mixed precision library Multiple Precision Floating-Point Reliably (MPFR) [55] is used for mixed precision arithmetic in MCALIB. Note that the bound on the value of t is not consistent with the original bound presented in [142] and detailed in Section 3.4.4. In the case of MCALIB the limitation on precision is determined by the limitations of MPFR which defines minimum and maximum precision vales of 2 and 256 respectively, bounding the working precision to $2 \leq W \leq 256$. The bound on the virtual precision has therefore been extended for MCALIB in order to allow simulation with zero non-random digits representing a complete loss of significance.

For MCA functions, FP values are converted to `mpfr_t` type variables. The `mpfr_t` type is a struct containing an arbitrary precision significand and a fixed precision exponent. The precision of the significand of any MPFR variable may be set independently at runtime to any value between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`, i.e. 2 and 256 respectively. For the purposes of MCALIB, the maximum precision required is $W_{max} = p + t_{max}$, which evaluates to 106 when using double precision

Input: Precision p FP operands x_f and y_f
Output: Precision p FP result r_f
x = extend(x_f, $p + t$);
y = extend(y_f, $p + t$);
r = extend(0.0, $p + t$);
x = inexact(x);
y = inexact(y);
r = mpfr_op(x, y);
r = inexact(r);
r_f = round(r, p);
return r_f

ALGORITHM 1: MCA Binary Operation

Input: Precision $p + t$ MPFR_T variable x
Output: Precision $p + t$ MPFR_T variable x (w. random perturbation applied)
if $x == 0$ **then**
    return x;
else
    ξ_f = (rand()/RAND_MAX) - 0.5;
    ξ = extend(ξ_f, $p + t$);
    ξ = mpfr_mul(pow(2, e_x - (t - 1)), ξ);
    x = mpfr_add(x, ξ);
    return x;
end

ALGORITHM 2: MCA Inexact Operation

operators, and the minimum required precision is $W_{min} = p + 0$, which evaluates to 24 when using single precision operators. Rounding in MPFR adheres to the C implementation of the Institute of Electrical and Electronic Engineers (IEEE)-754 standard and the default rounding mode **round to nearest even** is used for MCALIB.

The function for implementing MCA as per Equation 3.52 is shown in Algorithm 1. The FP operands are first converted to `mpfr_t` with precision W, and the result variable is initialized with the same precision. The random perturbation ξ is applied to the input operands using the inexact function shown in Algorithm 2. The arithmetic operation is then performed using an MPFR operation, rounded to W bits. Random rounding is then applied to the result using the inexact function, and the final result is then converted to its original format by rounding to p bits. MPFR

implements correct rounding according to the IEEE-754 standard with rounding error $\delta(x) \leq \epsilon$. Rounding error will occur both during the MPFR operation, δ_W, and when rounding to the original precision, δ_p. In order to implement correct rounding while simulating infinite precision during the MCA operation we must ensure that $\delta_W \leq \frac{1}{2}\delta_p$. The worst case scenario will occur when $t = 1$, as when $t = 0$ the initial MPFR rounding stage will round to the original precision with $\delta_W \leq \epsilon$ resulting in an exact value and $\delta_p = 0$. When $t = 1$ the rounding error in the MPFR operation will be limited as follows, assuming the general case $\delta \leq 2^{-p}$:

$$\delta_W \leq 2^{-(p+t)} \tag{4.1}$$

$$\leq \frac{1}{2}2^{-p},\ t \geq 1 \tag{4.2}$$

$$\leq \frac{1}{2}\delta_p \tag{4.3}$$

MCALIB implements the four basic arithmetic operations, $\{+, -, \times, \div\}$, unary minus, and the set of comparison operators, $\{==, !=, <, >, \leq, \geq\}$ for single and double precision formats. Comparison operators are implemented without using the inexact function in order to avoid changes to the branching behaviour of tested programs, and as such the comparison operators are implemented using extended precision MPFR operators only. The library includes two global parameters for controlling an MCA execution. The integer `MCALIB_T` sets the virtual precision, t, of MCA operations while the integer `MCALIB_OP_TYPE` allows the application of MCA to be controlled using a set of pre-processor symbols defined as part of the MCALIB library. These symbols, their values and their functions are shown in Table 4.1. Both parameters can be modified at runtime.

4.2.3 MCALIB Features & Workflow

MCALIB has been designed to facilitate the following analyses;

- Detection and quantitative analysis of sensitivity to rounding error.

MCALIB Control Symbols & Functions		
Symbol	Symbol Value	Function
MCALIB_IEEE	0	Disable MCA.
MCALIB_MCA	1	Enable MCA.
MCALIB_PB	2	Enable precision bounding only.
MCALIB_RR	3	Enable random rounding only.

Table 4.1: Name, values and function of MCALIB control symbols for parameter MCALIB_OP_TYPE

- Analysis of individual algorithms to determine if single or double precision FP arithmetic is required.
- Optimization of individual algorithms for precision.
- Comparison of algorithms to determine the most suitable implementation.

Each of these features is implemented by applying MCALIB to a problem according to the following MCALIB workflow as described below.

Algorithm Analysis

MCA is applied by first analysing the algorithm to be tested in order to determine the following:

- Where should MCA be enabled, i.e. which values are exact and which are not?
- What outputs are of interest, i.e. how is accuracy in this algorithm defined?

These questions are of high importance as they will have a significant impact on the results if not answered correctly. Although MCALIB provides an automated implementation of MCA, it is still a *naive* implementation, i.e. the system does not understand the difference between exact and inexact values and must be informed of this difference by the developer. Using MCALIB, all FP operations are re-written as function calls to the MCALIB library. Determining which inputs and outputs are to be treated as exact or inexact is a decision left to the developer, and is achieved by enabling or disabling precision bounding and random rounding individually as described below. Determining what outputs are of interest is a question of determining which variables determine the overall stability of an algorithm.

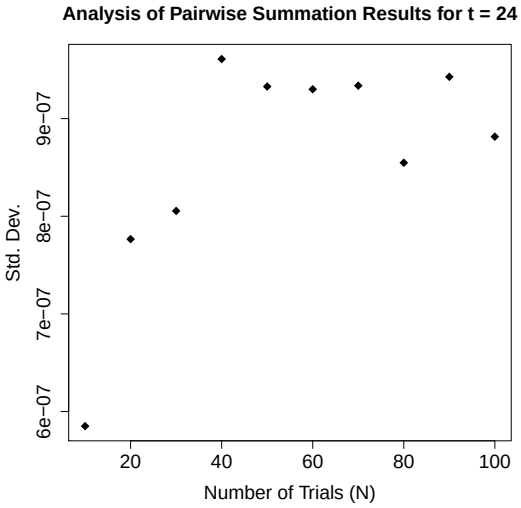


Figure 4.1: Pairwise summation - comparison of standard deviation for virtual precision $t = 24$ and different number of trials, N.

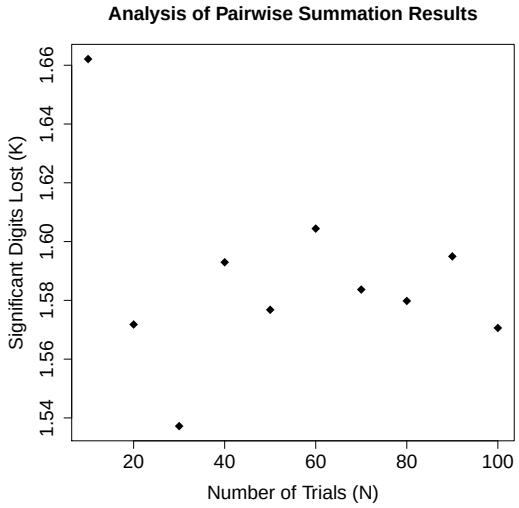


Figure 4.2: Pairwise summation - comparison of significant figures lost K and different number of trials, N.

Source Code Modification

Having determined the above, the second stage of the workflow involves modifying the source code. Implementation is a simple process and very few modifications are required. Developers need to add the MCALIB header file `mcalib.h` and modify their compilation process to utilize the cilly compiler and include the MCALIB library file `libmcalib.a`. MCA can be enabled or disabled where appropriate by setting the value of the control parameter `MCALIB_OP_TYPE` and the virtual precision can be set using the parameter `MCALIB_T`.

Data Collection

Once the original source code has been correctly modified the third stage of the MCALIB workflow is collection of data. In order to do this the following steps are required:

- Determine the input domain to be tested.
- Execute the required number of trials and collect data from the watched output(s).

As stated previously, MCALIB is a naive implementation of MCA and as such decisions regarding the input domain are left to the developer. This is an important step, as MCA performs a dynamic error analysis and results are only relevant to the input domain tested. For example, if testing a summation algorithm using uniformly distributed inputs, $x \in U[-1, 1]$, the results of MCALIB analysis will only be relevant for this domain. Once the input domain has been determined the trials must be executed and the output data collected. An important consideration for Monte Carlo methods is the number of trials to be performed, this number being directly affected by the sampling methodology in use. For the purposes of MCALIB, simplified random sampling has been implemented and it is recommended that a minimum of 100 trials be performed for any experiments. Decreasing the number of trials performed may have adverse effects on the results of analysis using techniques as shown in Section 4.3. As can be seen in Figures 4.1 and 4.2, decreasing the number of trials will adversely affect the results. The standard deviation and the calculated value of K do not converge until approximately $N = 50$. The recommended number of samples is based on experimental results presented in this chapter, worst case sample size considerations [76, Chapter 3], and experimental data presented in [142]. While the recommended number of samples may appear high, work presented in Chapter 6 will show that this figure can be reduced using techniques such as quasi-Monte Carlo Simulation. This will be the subject of future research.

Results Analysis

Having performed the required number of experiments and collected the relevant output data, the next stage of the MCALIB workflow is results analysis. Using the methods described in the next section, results of MCA trials may be analysed to determine the total number of digits lost to rounding error and the minimum precision required in order to avoid a total loss of significance. If no valid results are available then the virtual precision range should be widened, particularly at the top end, to collect more data at more stable precision values. If the normality tests fail

consistently, the developer should return to step 1 to re-analyse the algorithm and ensure that the input domain and outputs are being monitored correctly. Further testing may be performed in order to attempt to determine the source of rounding error reported by results analysis. Repeated testing may be performed with the type of random perturbation applied to operations is modified for each test. By comparing results of testing for full MCA analysis, (where random rounding and precision bounding are both applied), against results for random rounding and precision bounding alone, it is possible to determine whether a loss of significance is occurring due to rounding error, catastrophic cancellation, or a combination of the both.

4.3 Analysis of MCA Results

In previous publications [142] the analysis of MCA results has been limited to determining the number of significant digits, and pass/fail analysis performed by comparing the mean and standard deviation of MCA results. We feel that this approach can be expanded and more formally defined in order to provide a more rigorous definition of sensitivity to rounding error in MCA results, allowing analysts to draw more meaningful conclusions from the results of MCA analysis. In this chapter sensitivity to rounding error is defined using two measurements:

- The number of base-2 significant digits lost due to rounding error, K
- The minimum precision required to avoid a total loss of significance, t_{min}

We must first address the ideal case for error in MCA. If relative error is defined as in Section 2.4.2 then it has been noted that the relative error is limited by $\delta \leq 2^{-p}$ for binary FP systems [65, 82, 170]. From [142, page 19], the definition of relative error is used to determine the expected number of significant binary digits available from

a p-digit FP system:

$$\delta \leq 2^{-p} \tag{4.4}$$

$$p \geq -\log_2(\delta) \tag{4.5}$$

These definitions may be adapted for MCA by replacing the precision of the FP system, p, with the virtual precision, t, of an MCA operation. Thus the relative error δ in a MCA operation for a virtual precision t is given by $\delta \leq 2^{-t}$, and the expected number of significant binary digits in a t-digit MCA operation is at least t. Using this definition a proof has been provided [142, page 23] giving the total significant binary digits in a set of MCA results:

$$s' = \log_2 \frac{\mu}{\sigma} \tag{4.6}$$

Where μ is the mean and σ the standard deviation of the MCA results. Using the definitions in this section the total number of significant digits lost in a MCA result set, K, may be defined as follows:

$$K = t - s' \tag{4.7}$$

$$= t - \log_2(\frac{\mu}{\sigma}) \tag{4.8}$$

$$= \log_2(\Theta) + t \tag{4.9}$$

Where $\Theta = \frac{\sigma}{\mu} \rightarrow \mu \neq 0$ is the **relative standard deviation (RSD)** of the MCA results.

As noted by Sterbenz, [160, Chapter 7], in an ideal case a linear relationship exists between the precision of a FP system, p, and significant figures in the output. Using MCA, this linear relationship exists between t and $\log(\Theta)$. We identify the point of departure as when the algorithm being analysed is affected in a non-linear way by rounding error. We propose that the breakaway point in the linear model represents t_{min}; the minimum precision required to avoid a total loss of significance

in the results. In order to determine the best fit of the relative error model results below, outliers are not used in the calculation of K.

4.3.1 Linear Regression Analysis

In order to determine the value of t_{min} and K, a linear regression with a log transformed variable is used, with $\log(\Theta)$ as the dependent variable and t as the exploratory variable in the following form:

$$\log_{10}(\Theta) = \log_{10}(2^{K-t}) \tag{4.10}$$

$$= -\log_{10}(2)t + \log_{10}(2)K \tag{4.11}$$

$$= mt + c \tag{4.12}$$

where $m = -\log_{10}(2) = -0.30103$ is the slope and c is the intercept such that $K = \log_2(10^c)$. Due to the requirement of detecting outlying results, robust regression methods are used to evaluate the linear model. The example presented in [56] performs robust regression using M-Estimation through the iteratively re-weighted least squares (IRLS) approach for 2-D optimization. While this approach is ideal for MCA analysis due to its insensitivity to outliers, the approach can be simplified to a 1-D optimization problem as the slope of the linear model is already known. Given a set of MCA results for virtual precision values $t \in [1, t_{max}]$ a summary set is created by calculating Θ at each t value. It should be noted that while the samples used to calculate an individual value for Θ are independent and identically distributed (IID), the complete sample set is not in general identically distributed. Given these inputs the intercept c is calculated by minimizing the following objective function using Brent's method [14] for single variable optimization;

$$f(x) = \sum_{i=1}^{t_{max}} \gamma^{t_{max}-i} \rho_H(e_i) \tag{4.13}$$

where $e_i = \Theta_i - (mt_i + c)$ is the residual error, $c \in [(\Theta_{t_{max}} - mt_{max}) \pm 2m]$ is the initial search space for the intercept, $\gamma = 0.75$ and $\rho_H(e)$ is the Huber loss function [85];

$$\rho_H(e) = \begin{cases} \frac{1}{2}e^2, & \text{for } |e| \leq k \\ k|e| - \frac{1}{2}k^2, & \text{for } |e| > k \end{cases} \tag{4.14}$$

where $k = 1.345\sigma$ and σ is the standard deviation of the residual error set, e. Having determined the linear model, the outlying values of Θ are found by calculating a set of predicted values $P_t = mt + c$ and comparing these with the values for Θ obtained via MCA. If a value Θ_t differs from its equivalent predicted value, P_t, by more than half a binary digit it is classed as an outlier. The breakaway point, t_B is calculated by finding the highest t value where $|P_t - \Theta_t| > \log_{10}(2^{0.5})$. The value of t_{min} is then set to $t_B + 1$.

4.3.2 Assumption of Normality and Conditions on Results

In order to perform analysis using the statistical methods listed in this chapter the input data set is typically assumed to be normally distributed, however, in the case of MCA no assumption of normality is made. This is explicitly stated by Parker [142, p. 49] and is intended to allow for open-ended statistical testing of MCA results. In order to provide a strong estimate on the result of K and t_{min} the normality of the sample set must first be verified for each value of t. This is determined on the raw MCA data at each t step, requiring a total of $t_{max} - t_{min}$ tests. This is done using the Anderson-Darling test to assess the goodness of fit of the frequency distribution of results to a normal distribution. If the test fails, warnings are provided on the plotted output of the calculation, and the result sets that have failed the test are removed and not used for the calculation of K or t_{min}. The calculation of K and t_{min} must be done in conjunction with bounds on the input space of the function or algorithm under investigation, i.e. the results of the linear regression do not provide a guarantee of the error in an algorithm in the general case, but rather an estimate of the accuracy

of the algorithm under the specific conditions tested using MCALIB.

4.4 Testing & Case Studies

Testing is performed by varying the virtual precision, $1 \leq t \leq p$, and performing N executions at each t value. For the tests conducted in this chapter, unless stated otherwise, we use t values from 1 to 53 and number of trials at each t value $N = 100$. In this section we describe the programs used to test MCALIB.

4.4.1 Chebyshev Polynomials

Chebyshev polynomials [148] are a series of orthogonal polynomials typically used in approximation theory. In this case we have used Chebyshev polynomials of the first kind, defined as follows:

$$T_0(z) = 1 \tag{4.15}$$

$$T_1(z) = z \tag{4.16}$$

$$T_{n+1}(z) = 2zT_n(z) - T_{n-1}(z) \tag{4.17}$$

Polynomials of the first kind can be represented as unique polynomials satisfying the following trigonometric definition:

$$T_n(z) = \cos(n \cos^{-1}(z)) \tag{4.18}$$

Input: Vector $X[1...n]$
Output: Sum s of vector X
$n_{max} = 1$;
if $n \leq n_{max}$ **then**
    $s = X[1]$;
    for $i = 2$ *to* n **do**
        $s = s + X[i]$;
    end
else
    m = floor(n / 2);
    s = pw($X[1...m]$) + pw($X[m+1...n]$);
end
return s

ALGORITHM 3: Pairwise Summation Algorithm

Input: Vector $X[1...n]$
Output: Sum s of vector X
$s = 0.0$;
$c = 0.0$;
for i = *1 to* n **do**
    $y = X[i] - c$;
    $t = s + y$;
    $c = (t - s) - y$;
    $s = t$;
end
Return s

ALGORITHM 4: Kahan Summation Algorithm

In particular the $T_{20}(z)$ polynomial:

$$T_{20}(z) = \cos(20\cos^{-1}(z)) \tag{4.19}$$

$$\begin{aligned} &= 524288z^{20} - 2621440z^{18} + 5570560z^{16} \\ &- 6553600z^{14} + 4659200z^{12} - 2050048z^{10} \\ &+ 549120z^{8} - 84480z^{6} + 6600z^{4} \\ &- 200z^{2} + 1 \end{aligned} \tag{4.20}$$

has been analysed by both Wilkinson [170] and Parker [142], who note that due to catastrophic cancellation occurring among the coefficients of the expanded series, the polynomial becomes ill-conditioned at the roots near $z = \pm 1$.

4.4.2 Summation Algorithm

FP summation is a widely used operation that sums a sequence of n FP values:

$$s = \sum_{i=1}^{n} x_i, \text{ for } n \geq 3 \tag{4.21}$$

Due to its widespread use in algebraic operations, the accuracy of summation has been analysed in various publications and it has been shown that the relative error of the Naive summation algorithm grows with order $O(\epsilon n)$ [83, 114, 119]. For this chapter the Naive approach is compared with two alternative summation algorithms, the **Pairwise** [83] and **Kahan** [92] summation algorithms, shown in Algorithms 3 and 4. Both of these algorithms have been shown to reduce numeric instability. In the case of Pairwise summation this is done using a divide and conquer strategy that reduces the relative error to order $O(\epsilon \log n)$ while not increasing the number of arithmetic operations used. Kahan summation uses a compensated sum to track round-off error during summation and reduces relative error to order $O(\epsilon)$, but significantly increases the required number of arithmetic operations.

The Naive, Kahan and Pairwise sum methods are compared using a set of sample values generated using the following [154]:

$$x_i = 10^{-p} \tag{4.22}$$

$$p = \left\lceil \log_{10}(9i + 1) - 1 \right\rceil \tag{4.23}$$

for $1 \leq i \leq 1111$.

4.4.3 Linear Algebra

Linear algebra subroutines are widely used in computer science and engineering, and accurate implementation of these algorithms is essential. Their implementation necessitates a large number of numeric operations and MCA is well suited for

analysis of the potential effects of rounding error. For the purposes of this chapter we have tested two implementations for determining the solution to a dense $n \times n$ system of linear equations $Ax = b$.

The implementations used for testing are the linear equations software package (LINPACK) benchmark [43], a tool which uses Gaussian Elimination with partial pivoting as an example of a general engineering problem in order to test a systems peak performance in terms of floating point operations per second (FLOPS), and a standard implementation of LU decomposition with back substitution from Numerical Recipes [145]. Precision testing and error analysis have been performed using the array size $n = 100$ and the value of A and b set using the `matgen` method provided as part of the LINPACK implementation used in this test case [163]. Statistical measurements were performed using the Euclidean, (L^2), norm of the result vector $x[n]$, defined as follows:

$$||x|| := \sqrt{x_1^2 + ... + x_n^2} \tag{4.24}$$

4.4.4 L-BFGS Optimization

The limited memory BFGS (L-BFGS) optimization method [115] is an implementation of quasi-Newton optimization using the Broyden, Fletcher, Goldfarb, Shanno (BFGS) update method for approximation of the Hessian Matrix. L-BFGS stores a finite number of vectors to represent the approximation, unlike the original BFGS method which stores a dense $n \times n$ approximation. An important part of this algorithm is the line search method, used to determine the local minimum $x*$ of an objective function $f : \mathbb{R}^n \to \mathbb{R}$. The objective function used for testing in this chapter is the Rosenbrock function [151], a well known convex function used for performance testing of optimization systems. This function has been provided as part of the L-BFGS implementation used for this paper [115], and is implemented for 10 dimensions

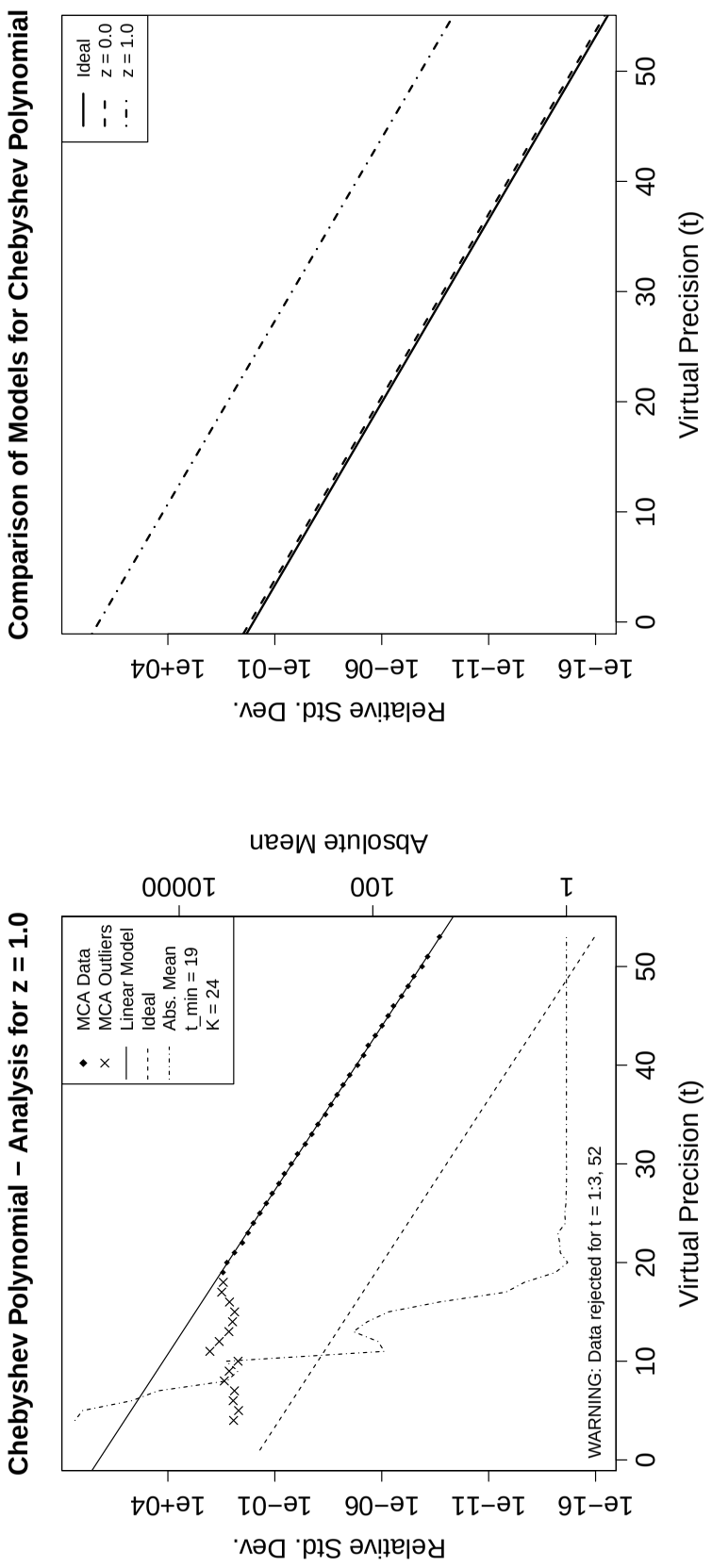
using:

$$f(\mathbf{x}) = \sum_{i=1}^{10}[(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2], \forall x \in \mathbb{R}^n \tag{4.25}$$

with the input vector x defined as follows;

$$x[i] = \begin{cases} 1.2 & \text{if } i \text{ is odd} \\ 10 & \text{if } i \text{ is even} \end{cases} \tag{4.26}$$

for $i \in [1, 10]$. The L-BFGS implementation used for testing provides a choice between four different line search methods, Moore-Thuente, Armijo, Wolfe and Strong Wolfe [40, 128] methods. Testing has been conducted for all four line search methods and statistical measurements are again performed using the Euclidean norm of the result vector.



(a) Chebyshev Polynomial - Sensitivity to rounding error at $z = 1.0$

(b) Chebyshev Polynomial - Comparison of results for $z = 0.0$ and $z = 1.0$

Figure 4.3: MCA analysis of Chebyshev Polynomial

Results - Chebyshev Polynomial		
Input - z	**Min. Req. Precision** - t_{min}	**Sig. Fig. Lost** - K
0.0	5	0.5
0.2	5	5.4
0.4	11	11.5
0.6	13	15.2
0.8	18	20.0
1.0	19	24.0

Table 4.2: Full Analysis of Chebyshev Polynomial

4.5 Results

In this section we present results of MCA analysis of several sample algorithms. Throughout this section results of MCA analysis are presented using plots generated via methods described in Section 4.3. The plots shown in Figures 4.3(a), 4.5(a), 4.6(a), and 4.7(a) provide detail on the results of the linear regression analysis. These compare the linear model with the ideal error case, ($\delta = 2^{-t}$), the experimental MCA results which were classified as outliers are clearly marked, as well as a plot of the absolute mean, $|\mu|$ to allow the mean to be checked in case it approaches zero. The plots are designed to provide a method for quick visual inspection of the MCA results. Inside the legend the magnitude of K, indicated by the distance between the linear model and the ideal case, and the value of t_{min}, indicated by the position of the outlying data points, are given. The second type of plot presented, (Figures 4.3(b), 4.5(b), 4.6(b), and 4.7(b)), is designed to provide a comparison of the different algorithms being tested. These plots compare the linear models generated through analysis of the MCALIB results with the ideal error case.

4.5.1 Error Detection and Optimization of Sample Algorithms

One of the primary functions of MCA is to detect sensitivity to rounding error within tested algorithms, indicated by a large variance in the results of repeated executions. Using the relative error model and the methods detailed in Section 4.3, it is possible to determine the overall sensitivity of tested algorithms to rounding error and to

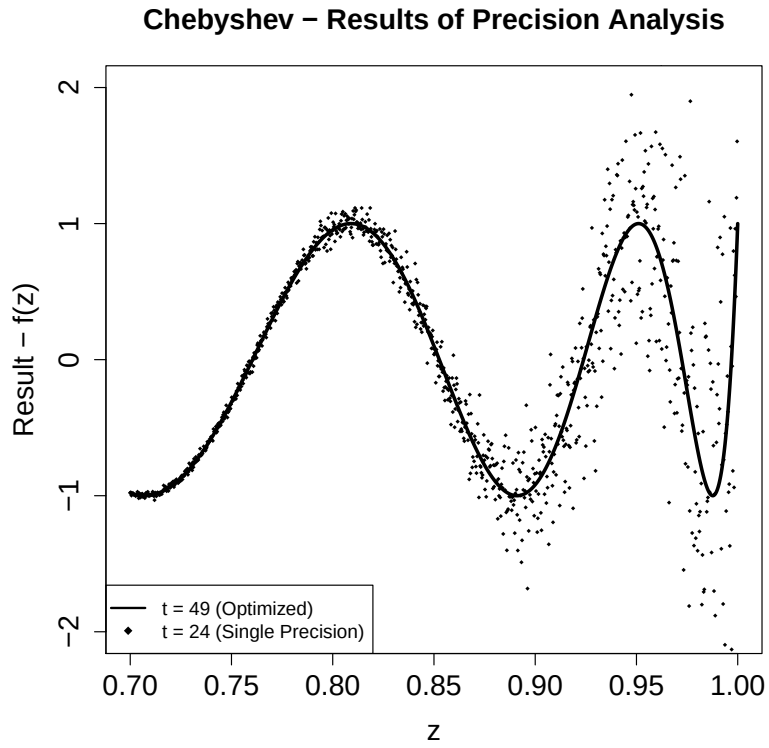


Figure 4.4: Chebyshev Polynomial - Comparison of single ($t = 24$) and optimized ($t = 49$) precision.

Results - Chebyshev Polynomial			
Type	t	μ	Θ
Single	24	0.9985	1.2119e+00
Optimized	49	1.0000	3.4492e-08

Table 4.3: Comparison of Single and Optimized Precision Results for Chebyshev Polynomial (using $z = 1.0$)

optimize these algorithms by determining their minimum precision requirements.

(a) L-BFGS Optimization - Analysis of More-Thuente line search method

(b) L-BFGS Optimization - Comparison of More-Thuente & Wolfe line search methods

Figure 4.5: MCA analysis of L-BFGS Optimization methods.

Analysis of L-BFGS Optimization		
Search Type	**Min. Req. Precision - t_{min}**	**Sig. Fig. Lost - K**
More-Thuente	48	8.7
Wolfe	19	8.9
Str. Wolfe	36	8.8
Armijo	53	8.9

Table 4.4: Analysis of Line Search Methods for L-BFGS Optimization

For the Chebyshev Polynomial, testing has been conducted using input values for z between 0 and 1 in steps of 0.2, conducting $N = 100$ executions for all t values between 1 and 53 at each z step. Results for all cases are shown in Table 4.2, results for the worst case $z = 1$ are detailed in Figure 4.3(a) and results for the $z = 0.0$ and $z = 1.0$ cases are compared in Figure 4.3(b). Initially at $z = 0$ the sensitivity to rounding error is negligible, as evidenced by a low value for t_{min} and less than 1 significant figure lost to rounding error. As z is increased to approach the root at $z = 1$, the number of significant figures decreases until at the worst case point, $z = 1$, 24 significant figures are lost to rounding error. At this point the minimum precision required to avoid a total loss of significance in the results has risen to 19 bits. Having quantified the sensitivity to rounding error for input values between 0 and 1, it is possible to use the values for K and t_{min} to optimize this algorithm and determine the precision level required to achieve results normally expected from single precision FP operators. Previously this was often achieved by simply switching to double precision FP operators. MCALIB allows for the effects of rounding error to be quantified, and for this information to be used to determine a required precision level. This can be done by simply adding the expected number of digits lost to the required precision level, 24 in this case, and ensuring the resulting value is greater than or equal to t_{min}. Table 4.2 shows that a precision of at least 19 bits is required, and due to the expected loss of significant figures for the worst case input, $K = 24.02$, a precision of $\lceil p + K \rceil = 49$ is required. The results of comparison testing between t values of 24 (single precision), and 49, (optimized precision), are shown in Table 4.3. These results have been produced using the worst case input, $z = 1$. It can

be seen that the relative standard deviation is 10^8 times lower for the optimized case, and is the same order of magnitude as the maximum relative error expected from single precision arithmetic, ($\delta = 2^{-24} \approx 6 \times 10^{-8}$). Figure 4.4 plots the results of the Chebyshev polynomial for both single ($t = 24$) and optimized ($t = 49$) precision calculated using MCALIB. From this plot the difference between the two precision levels can be seen. A precision level of 49 results in a smooth curve, while using a level of 24 results in a random spread of points.

(a) Summation Algorithm - Analysis of Pairwise Summation Method

(b) Summation Algorithm - Comparison of Results for Pairwise and Naive Algorithms

Figure 4.6: MCA analysis of Summation methods.

Analysis of Summation Algorithms		
Algorithm Type	**Min. Req. Precision - t_{min}**	**Sig. Fig. Lost - K**
Naive	7	7
Kahan	7	7
Pairwise	1	1.6

Table 4.5: Summation Algorithm Results - Naive, Kahan & Pairwise

4.5.2 Comparison of Single and Double Precision Floating Point Formats

A simpler form of error analysis that may be performed with MCALIB is the comparison of single and double FP operators. In this case an individual algorithm may be tested in order to determine if the single precision FP format is sufficient for the given input domain, or if double precision type operators are required. This type of analysis has been used to determine the sensitivity to rounding error of different line search algorithms as used in L-BFGS optimization of the n-dimension Rosenbrock function, allowing for both the comparison of line search methods and the selection of single or double precision operators for the tested input domain. The results of error analysis for all four line search methods are shown in Table 4.4. The results of testing the More-Thuente line search are plotted in Figure 4.5(a), and results for the More-Thuente and Wolfe line search methods are compared in Figure 4.5(b). From the results table it can be seen that all four line search methods lose approximately nine significant figures to rounding. This result coupled with the results for t_{min} indicates that single precision FP operators are insufficient for that algorithm, however, it can be seen from the warning on the bottom left of Figure 4.5(a), a total of 47 data points have been rejected due to non-normality of the data set. This is most likely caused by the iterative nature of the algorithm under investigation, and the fact that the optimization process is attempting to find a solution within an error bound of 2^{-53}. Given that the virtual precision of the MCA operators is varied between 1 and 53 the error analysis method is having an adverse affect on the accuracy of the solution. For the purposes of demonstration the non-normal data points have been forcefully included in the results analysis, but in practice these results are not viable

and the experimental conclusions should be rejected. As such, while these results indicate the possibility that single precision FP is not suitable for the tested input domain, further analysis is required. Given the adverse effect of the analysis type on the results of the algorithm, and the inability to perform statistical analysis on the results of analysis, it is clear that MCA analysis and as an extension MCALIB is not suited to analysis of this algorithm and potentially algorithms of this type. Given that statistical analysis is not possible it is likely that static analysis methods, or potentially the use of more robust optimization methods such as iterative refinement, would be required in order to perform analysis.

(a) Analysis of LINPACK benchmark

(b) Comparison of Linear Solvers

Figure 4.7: MCA analysis of LINPACK benchmark

Comparison of Linear Solvers		
Algorithm Type	**Min. Req. Precision** - t_{min}	**Sig. Fig. Lost** - K
LU Decomp. w. Back Sub.	17	7.1
LINPACK	17	7.3

Table 4.6: Linear Solvers - Comparison of LINPACK and LU Decomposition with Back Substitution

4.5.3 Comparison of Algorithm Implementations

In addition to performing an analysis of individual algorithms demonstrated in the previous section, MCALIB can be used to compare competing algorithms or implementations in order to determine the best approach. The first set of algorithms tested are algorithms for FP summation, including the Naive, Kahan and Pairwise algorithms. The results of analysis for all three algorithms are shown in Table 4.5, the results of analysis of the Pairwise method are detailed in Figure 4.6(a) and the results for the Pairwise and Naive methods are compared in Figure 4.6(b). From these results it can be seen that all three algorithms demonstrate low sensitivity to rounding error. The Pairwise method demonstrates significantly lower sensitivity to rounding errors when compared with the alternative methods. This is evident in the lower value for t_{min}, with a result of 1 for the Pairwise algorithm versus 7 for the Kahan and Naive methods. The Pairwise method is also losing less than 2 significant digits to rounding error, compared with the 7 significant digits lost for the Naive and Kahan methods. While all three methods demonstrate low sensitivity to rounding error and may be analysed using single precision operators, the Pairwise method provides the best approach for FP summation for the tested input domain, (as detailed in Section 4.4.2).

This same type of analysis has also been used to compare a linear solver from Numerical Recipes [145] with the one in the LINPACK benchmark [43]. The results for analysis of the two algorithms are shown in Table 4.6, the results of analysis of the LINPACK benchmark are detailed in Figure 4.7(a) and results for both implementations are compared in Figure 4.7(b). As was the case with the summation algorithms, both algorithms show a low level of sensitivity to rounding error and the

MCALIB				CADNA	
Input z	K	t_{min}	**MFLOPS**	**Sig. Fig. Lost**	**MFLOPS**
0.0	5	0.5	1.48	0	5.54
0.2	5	5.4	1.45	1	5.39
0.4	11	11.5	1.45	2	5.40
0.6	13	15.2	1.50	4	5.52
0.8	18	20.0	1.53	5	5.52
1.0	19	24.0	1.37	6	5.49

Table 4.7: Chebyshev polynomial- Comparison of MCALIB & CADNA

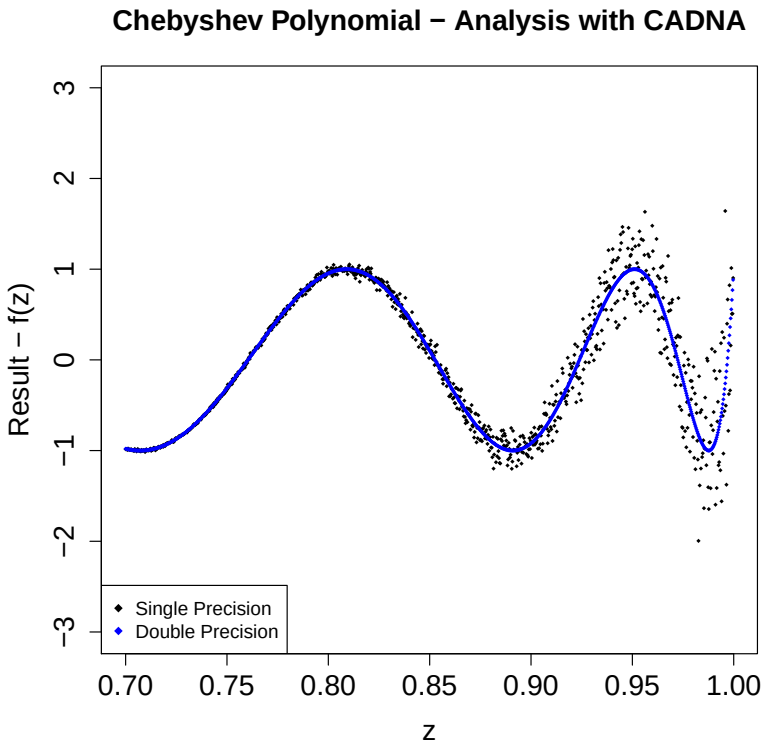
result for t_{min} for both methods indicates that single precision formats are suitable for use with the tested input domain.

The error analysis results also clearly indicate a similar level of sensitivity to rounding error available in both algorithms, this being demonstrated by the approximately seven significant figures lost to rounding error in both cases. The overall effect of rounding error on the results for the LINPACK benchmark can be seen in Figure 4.7(a). As the virtual precision is increased beyond $t = 17$ the relative standard deviation decreases exponentially, forming a linear relationship with the virtual precision. These results can also be produced using single precision FP operators if necessary. However, the values for t_{min} and K indicate that the algorithm becomes highly sensitive to rounding error if the precision is decreased below 17. Furthermore if the required significance of the results must be equivalent to single precision FP, a precision of $\lceil p + K \rceil = 32$ is recommended when using these algorithms on the tested input domain.

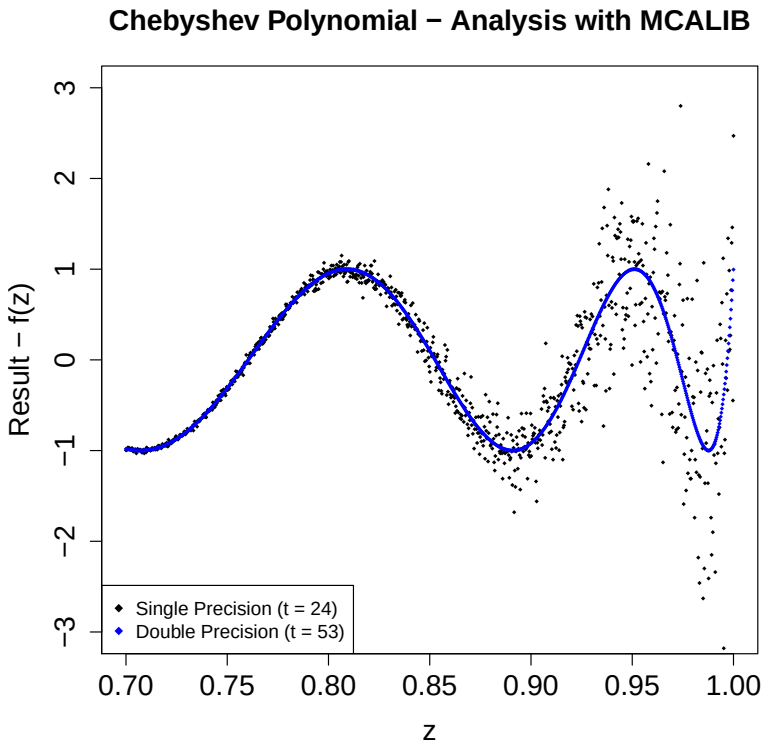
4.5.4 Comparison of Results with Existing Methods

Analysis of the Chebyshev polynomial test case has been performed using control of accuracy and debugging for numerical applications (CADNA), an existing analysis scheme implement discrete stochastic arithmetic (DSA), in order to compare MCALIB the state of the art. For the purposes of this section both the error analysis results and the results of performance testing have been provided. Testing

with CADNA has been conducted using the same methodology as in the preceding subsection, the input value z is increase from $z = 0$ to $z = 1.0$ in steps of 0.2 using the CADNA double precision operator. The results of analysis with CADNA are compared with the results generated using MCALIB in Table 4.7, along with performance results for both systems presented in FLOPS. Note that MCALIB reports a significantly higher number of significant figures lost to rounding error for all input values except $z = 0$, the input least susceptible to rounding error. This difference is due, at least in part, to the difference in the way random perturbations are handled by the different systems. In the case of systems like CADNA which implementDSA, randomness is applied by varying the rounding mode, essentially perturbing a single digit, even in cases where cancellation results in more than on non-significant digit on the significand of the result. This is avoided using MCALIB, where all non-significant digits are perturbed. The net result is that the the random perturbations applied with MCALIB can be up to twice as large as those applied by rounding methods alone. This effect is further demonstrated in Figures 4.8(a) and 4.8(b). In this figure a total of 1000 data points have been taken for input values between $z = 0.7$ and $z = 1.0$, using single and double precision CADNA operators for figure 4.8(a) and virtual precision values $t = 24$ and $t = 53$ for figure 4.8(b). Note that in the single precision case the distribution of results for MCALIB is significantly larger than that of CADNA. The drawback in the case of MCALIB is evident when comparing the performance results listed in table 4.7. The speed of the CADNA implementation has been measured for this case as approximately 5.5 MFLOPS, while the performance of MCALIB is less than half that value at approximately 1.5 MFLOPS. In addition analysis with CADNA only requires 3 trials compared to the 100 required by MCALIB, taking this into account the analysis performed for this section may be completed more than 100 times faster using CADNA.



(a) Chebyshev Polynomial - Comparison of single and double precision operators using CADNA



(b) Chebyshev Polynomial - Comparison of single and double precision operators using MCALIB

Figure 4.8: Analysis of the Chebyshev polynomial with Monte Carlo Arithmetic (using MCALIB) and the Discrete Stochastic Arithmetic (using CADNA).

4.6 Summary

The MCP system presented in this chapter provides developers and numeric analysts a tool set for quantifying the effects of rounding error on the output of a program for a specified input domain. Using this system the first of the two approaches to rounding error presented in section 2.6.5 is advocated for the adoption of error analysis techniques as part of the software development life cycle (SDLC). In order to achieve this MCP attempts to provide functionality satisfying the needs of both developers - seeking high level, automated analysis of existing programs, and numeric analysts - seeking extendible, in depth analysis of FP formats and operators. It is in this vein of thinking that MCP has been developed. Its application is facilitated by MCALIB, an open source tool which applies source-to-source compilation to rewrite FP operators to call our MCA library. Furthermore, analysis techniques for better interpretation of MCA results have been presented. Using this methodology sensitivity to rounding error is quantified with two measurements:

- K - the number of significant figures lost to rounding error.
- t_{min} - the minimum precision required to avoid total loss of significance.

Both values are measured via analysis of the linear relationship between the RSD and virtual precision of a set of MCA results and are determined with a novel approach utilizing robust linear regression methods. The analysis technique expands the use of MCA and further demonstrate the benefit of this type of analysis for evaluating FP SW. Further work in this area will focus on investigating the use of quasi Monte Carlo techniques to reduce the required number of trials, and the use of MCA analysis to facilitate mixed precision implementations.

Chapter 5

FPGA-based Floating Point Unit for Rounding Error Analysis

5.1 Introduction

Monte Carlo arithmetic (MCA) is typically performed using software (SW) routines and as such its implementation involves a drastic reduction in performance. Systems utilizing field programmable logic (FPL) offer a platform in which hardware (HW) acceleration can be applied to arbitrary algorithms. In this chapter we describe a processor/co-processor system capable of performing virtual precision MCA for single precision floating point (FP) operators using a Xilinx Zynq system on a chip (SoC) for implementation and testing [178, 179]. After performing initial testing of the co-processor system using the linear equations software package (LINPACK) benchmark to determine system performance, the co-processor has been customized using high level synthesis (HLS) methods in order to improve system performance for the specific case of the LINPACK benchmark. The chapter is organized as follows; in Section 5.2 the system implementation - including implementation of MCA operators, the Zynq platform, HW/SW co-design and co-processor implementation is detailed. Section 5.3 presents initial results of testing the co-processor implementation including performance and system logic utilization. The customization of

the co-processor along with revised results is presented in Section 5.4, and finally conclusions are presented in Section 5.5.

5.2 System Implementation

The MCA floating point unit (FPU) is implemented as an field programmable gate array (FPGA) co-processor using a Xilinx Zynq Z-7020 SoC. The platform contains both an ARM Cortex-A9 MPCore Processor, referred to as the processing system (PS), and an Artix 7 FPGA, referred to as the programmable logic (PL) [37], this architecture is detailed in Figure 5.1. The combination of the processor core and the FPGA fabric within a single device allows for simplified implementation of a combination processor/co-processor system. In the case of the MCA FPU, SW executed on the ARM processor handles control and data input/output (IO) with all FP operations passed to the co-processor via an advanced extensible interface (AXI) Lite bus. The MCA FPU core was developed using the high-level C-to-RTL design SW Vivado HLS (http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html). Using Vivado, the MCA FPU is described using standard C statements and during synthesis and implementation, FP operations are translated into a set of FP modules based on the Institute of Electrical and Electronic Engineers (IEEE)-754 FP library.

5.2.1 MCA Operator Implementation

The MCA FPU is able to perform five arithmetic operations; add, subtract, multiply, divide and unary negative. The arithmetic operations are implemented by coupling Xilinx LogiCORE FP operators [174] with control logic, a configuration register, a set of random number generators and a set of perturbation modules implementing the inexact function. The basic architecture and data-flow of the co-processor is shown in Figure 5.2. In order to implement MCA functionality the precision of the FP operations must be extended in order to accommodate the random number to be

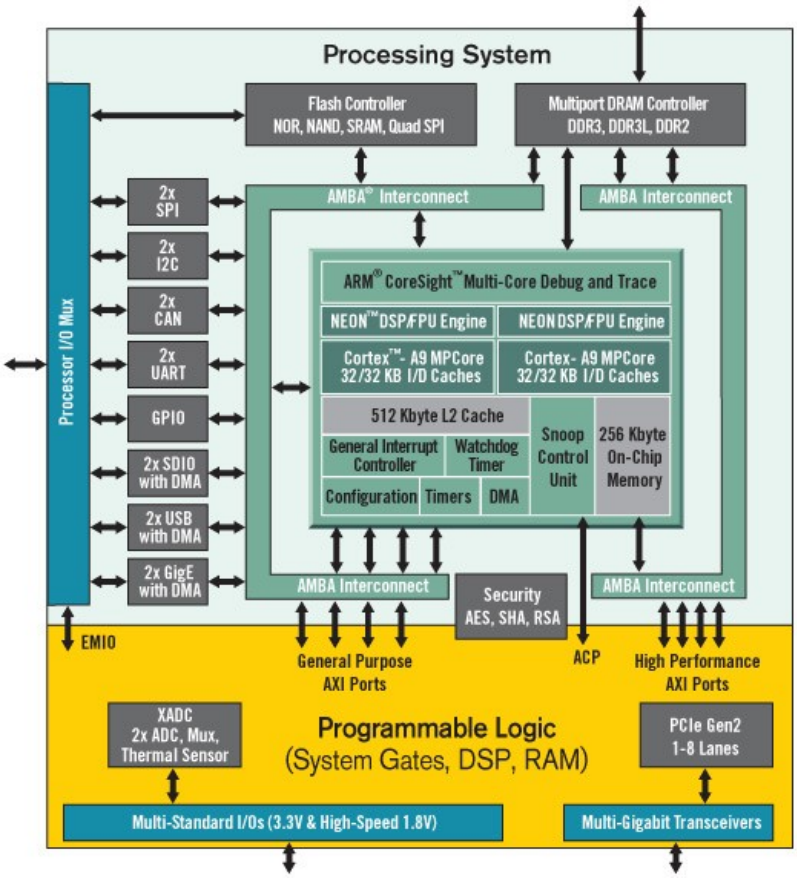


Figure 5.1: Zynq SoC architecture block diagram [87]

appended to the significand. As it is not feasible to modify the internal structure of the Xilinx LogiCORE, double precision FP cores are used and the virtual precision of the MCA operations is limited to $0 \leq t \leq p$, where $p = 24$ is the precision of single precision FP operations. Thus the double precision FP operators are used to implement single precision MCA operations. Random numbers are generated using maximally equi-distributed combined Tausworthe generator (MECTG) [106]. The configuration information contains both an opcode defining which operation is to be performed and the value of the virtual precision, t. The configuration register is a 32-bit register, with the most significant 16 bits reserved for the opcode, and the least significant 16-bits reserved for the virtual precision. IO transfer to and from the co-processor is handled using an AXI4 Lite interface [177]. Using this interface the 3 FP operands and the configuration information are stored as

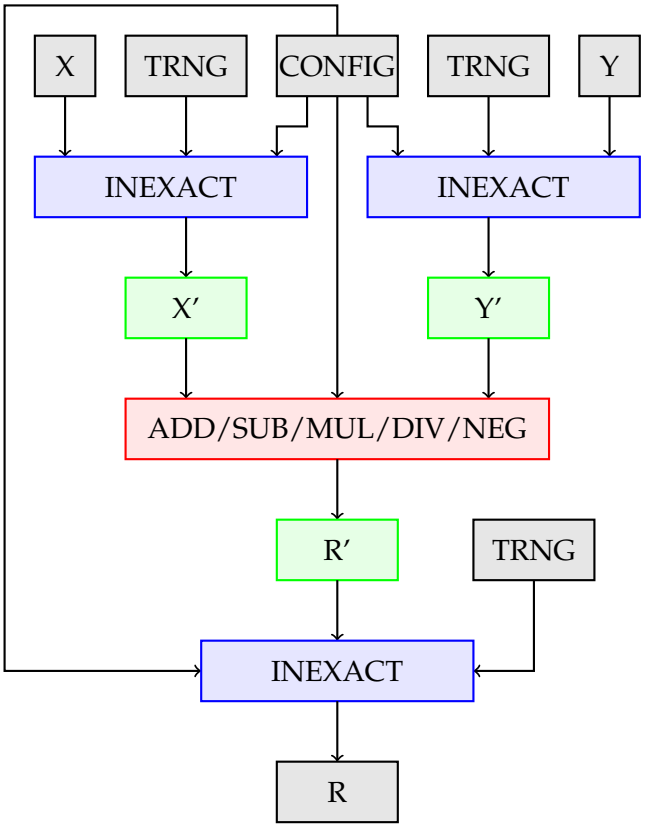


Figure 5.2: Procedure for MCA FPU operations

registers within the co-processor architecture and can be read and written as needed.

MCA operations are performed in terms of the inexact function as detailed in Equations 3.50 and 3.52 in Section 3.4.1;

$$r = x \circ y \tag{5.1}$$

$$= \text{round}(\text{inexact}(\text{inexact}(x) \circ \text{inexact}(y))) \tag{5.2}$$

where $\circ$ represents an arithmetic operation $\{\times, +, -, \div\}$. For the purposes of the co-processor implementation, the unary negative operator is implemented using a subtraction operation where an input x is negated to form the result r by:

$$r = 0 - x \tag{5.3}$$

This implementation may be used without performance penalty as the FP value 0 is considered to be exact under MCA and the inexact function is not applied, furthermore the constant value is not transferred over the bus interface. An individual MCA operation is performed according to the flowchart in Figure 5.2 as follows:

1. The FP inputs x and y are separated into individual sign, exponent and significand components labelled s_x, e_x, m_x and s_y, e_y, m_y respectively. Rounding up from single to double precision format is performed, (if necessary) in SW executed by the PS due to the 64-bit bus width. The values are separated using the following bitwise operations:

$$s = (x \gg 63) \tag{5.4}$$

$$e = (x \gg 52)\ \&\ 0\text{x}7\text{ff} \tag{5.5}$$

$$m = (e == 0)\ ?\ (x\ \&\ (2^{52} - 1)) : (x\ \&\ (2^{52} - 1))\ ||\ (2^{52}) \tag{5.6}$$

2. Two random values ξ_x and ξ_y are generated by the MECTGs. Each value must be in the range $[0, \frac{1}{2})$ and will be either added or subtracted to the FP operand with a 50% probability. As such each value is input to the inexact function as a 24-bit signed fixed point value giving a total range of $(-\frac{1}{2}, \frac{1}{2})$.
3. The input operands x and y are perturbed using the inexact function:

$$x' = \text{inexact}(x, \xi_x, t) \tag{5.7}$$

$$y' = \text{inexact}(y, \xi_y, t) \tag{5.8}$$

The inexact function is performed by first aligning the value ξ with significand and then shifting the random value to the right by t digits. As the significand is being stored as a normalized fixed point value, this would normally only require shifting the value of ξ right by $t + 1$ places, however the extended precision must be accounted for and furthermore, as single precision functionality is being simulated using double precision FP operators, the extra five

bits in the signifcand must also be accounted for - (Note: only five bits are considered *extra* as MCA requires the precision to be extended to up to $2p$, as such $53 - (2 * 24) = 5$). All together this results in a right shift of $t + 1 - 24 - 5$ or $\xi = \xi \ll (28 - t)$. Having aligned and shifted the random value it may now be added/subtracted to the significand using fixed point operators. If the results of the operation is negative the sign of the FP value is adjusted and the significand set to a positive value. Once this operation has been performed the new value of the signficand must be correctly normalized and the exponent of the original FP adjusted accordingly. There a four possible scenarios for the normalization stage:

(a) **The significand is already normalized:** In which case, no further action is taken.

(b) **The significand is equal to zero:** In which case, the sign and exponent are set to zero in order to correctly encode FP $+0$.

(c) **The significand is too large:** If the significand is too large, (i.e. $m \geq 2^{53}$), then it must be shifted to the right and the value of the exponent increased accordingly. Due to the range of the inputs to the fixed point operation the range of possible outputs is limited by $0 \leq m \leq 2^{54} - 1$ and as such the largest right shift and adjustment that will be required is one. The exponent must be checked to determine if $e = e_{max}$ in which case overflow has occurred and the result is $\pm\infty$. If $e < e_{max}$ then the exponent and significand are adjusted as follows:

$$e = e + 1 \tag{5.9}$$

$$m = m \gg 1 \tag{5.10}$$

(d) **The significand is too small:** The significand is deemed too small when in the range $0 \leq m \leq 2^{52} - 1$. In this case the significand must be shifted to the left and the exponent adjusted accordingly. The size of the shift is

determined by the number of leading zeros, λ, and due to range of possible values of the significand the number of leading zeros is in the range $1 \leq \lambda \leq 24$. The number of leading zeros is determined using a leading zero detector (LZD) optimized for very large scale integration (VLSI) systems that determines the number of leading zeros in $\log(n)$ stages [137]. Once the number of leading zeros is determined the significand is shifted to the left by λ places and the exponent decreased, unless the number of leading zeros is greater than the value of the exponent, in which case gradual underflow is occurring and the exponent is set to zero:

$$m = m \ll \lambda \tag{5.11}$$

$$e = \begin{cases} 0 & \text{if } e < \lambda \\ e - \lambda & \text{if } e \geq \lambda \end{cases} \tag{5.12}$$

4. Once the perturbed values x' and y' are calculated they are re-assembled into FP values by recombining the individual sign, exponent significand values:

$$x = (s \ll 63) \;||\; (e \ll 52) \;||\; (m \;\&\; (2^{52} - 1)) \tag{5.13}$$

5. Once the FP values are re-built the initial result, r' is computed using standard FP operators:

$$r' = x' \circ y' \tag{5.14}$$

6. The initial result r' is now disassembled into separate values for sign, exponent and significand using the procedure in step 1.
7. The random value ξ_r is generated using a third MECTG according to step 2.
8. The value of the final result is determined by perturbing the initial result:

$$r = \text{inexact}(r', \xi_r, t) \tag{5.15}$$

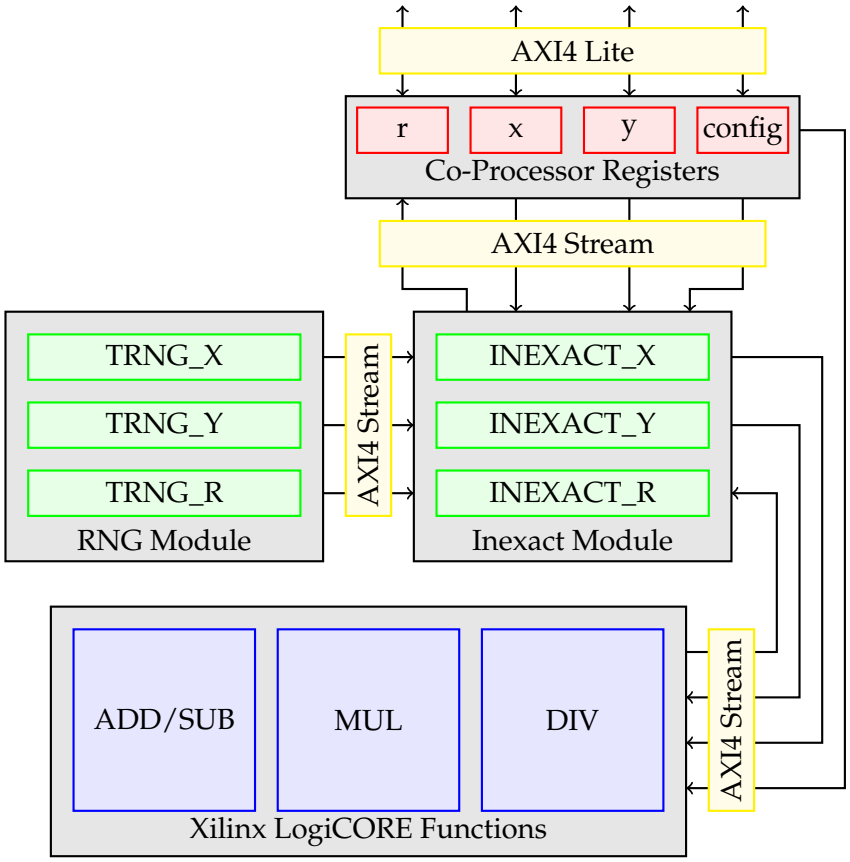


Figure 5.3: System Overview of the MCA FPU co-processor

according to step 3.

9. The final result r is reassembled according to step 4.

5.2.2 Co-Processor Implementation

The co-processor is implemented using the C to register transfer level (RTL) SW Vivado HLS. Using this SW the MCA operators are described using C syntax, along with descriptions of the IO protocol implemented using an AXI4 Lite bus interface. The complete co-processor architecture is shown in Figure 5.3, this implementation is based on the protocol processing methodology presented in [99]. Using this implementation method internal data connections between functional units are implemented using AXI4-Stream connections allowing for pipelined data-flow to be implemented at a later stage if required. Synthesis of the co-processor has been

Vivado HLS Settings	
Vivado Version	2014.4 Build 1071461
Target Device	xc7z020clg484-1
FPGA Board	Xilinx ZC702 Evaluation Board
Target Clock Period	10 ns
Target Clock Uncertainty	0 ns
Simulator	Vivado Simulator
RTL Simulation Selection	Verilog
RTL Export Format	IP Catalogue (w. Verilog Evaluation).

Table 5.1: Vivado HLS Synthesis, Co-Simulation and RTL Export Parameters

Timing Results (Estimated)		
Max Frequency	116 MHz	
Clock Uncertainty	1.25 ns	
Latency - Add/Subtract	15	
Latency - Multiply	16	
Latency - Divide	41	
Resource Utilization (Estimated)		
block RAM (BRAM)	0 / 280	0 %
digital signal processing/processor (DSP)	14 / 220	6 %
flip flop (FF)	6376 / 106400	5 %
look up table (LUT)	13964 / 53200	26 %

Table 5.2: Vivado HLS Synthesis Results - MCA FPU Co-Processor

performed targeting the Xilinx ZC702 Evaluation kit using settings as shown in Table 5.1. Initial results from synthesis are shown in Table 5.2 and initial synthesis of a IEEE-754 FPU are shown for comparison purposes in Table 5.3. These results indicate an expected overall increase in logic utilization of approximately 80% over an equivalent IEEE-754 FPU implementation. The performance results of the complete system will largely depend on the level of overhead associated with data transfer from the PS to the co-processor, and to a lesser extent the ratio of operation types used, however it is possible to estimate the throughput of the MCA co-processor based on the timing results presented in Table 5.2. These results indicate a minimum speed of 2.8 Mfloating point operations per second (FLOPS), a maximum speed of 7.7 MFLOPS and an average speed of 6.4 MFLOPS is theoretically possible.

Timing Results (Estimated)		
Max Frequency	116 MHz	
Clock Uncertainty	1.25 ns	
Latency - Add/Subtract	4	
Latency - Multiply	5	
Latency - Divide	30	
Resource Utilization (Estimated)		
BRAM	0 / 280	0 %
DSP	14 / 220	6 %
FF	4728 / 106400	4 %
LUT	6541 / 53200	12 %

Table 5.3: Vivado HLS Synthesis Results - IEEE FPU Co-Processor

5.2.3 Processor Implementation

The MCA FPU co-processor is connected to a Zynq7020 SoC containing the Arm Cortex A9 central processing unit (CPU). This processor/co-processor interface has been implemented using the Xilinx LogiCORE IP Processing System 7 v4.00.a, the SW interface built around the Zynq PS [175]. The PS has been implemented as standard for the Zynq7020 system with the general purpose AXI master interface, `M_AXI_GP0` enabled for the purposes of connecting the PS to the AXI interconnect and subsequently to the MCA FPU. Clock signals to the co-processor are implemented with the PL fabric clocks generated by the PS and have been set to a frequency of 100 MHz based on the estimated timing results from Vivado HLS. The LogiCORE IP AXI Interconnect v2.1 [172] is included to connect the PS to modules with AXI type interfaces, and is implemented with a single AXI Lite slave interface, connected to the PS, and two AXI Lite master interfaces, connected to the MCA FPU and to a AXI Timer. The timer module is included for the purposes of measuring performance using a LogiCORE IP AXI Timer v2.0 [173], and is setup for a 32-bit timer/counter in polling mode. The final module included in the system is the LogiCORE IP Processing System Reset Module v5.0 [176], which implements synchronous reset signals for the PS and associated peripherals on the PL. For the purposes of synthesis and implementation Vivado default settings have been used. A summary of Vivado's

Resource Utilization			
Type	**Used**	**Available**	**Utilization**
Slice LUT	8256	53200	15.5%
Slice Registers	6405	106400	6%
F7 Multiplexers	7	26600	$< 1\%$
F8 Multiplexers	1	13300	$< 1\%$
Slices	2456	13300	18.5%
LUT as Logic	7578	53200	14%
LUT as Memory	678	53200	1.3%
LUT/FF Pairs	8971	53200	17%
BRAM	0	140	0%
DSP	14	220	6.5%

Table 5.4: Vivado Implementation Results - MCA FPU Co-Processor

post implementation resource utilization report is presented in Table 5.4. Note that the summary presents results of logic utilization for the MCA FPU co-processor only and does not include resources used by peripheral overhead such as the interconnect, timer or reset modules which are also implemented on the PL.

5.3 Testing & Results

5.3.1 The LINPACK Benchmark

Linear algebra routines are widely used in science and engineering and accurate implementation of these algorithms is essential. Their implementation necessitates a large number of numeric operations and MCA is well suited for analysis of the potential effects of rounding error. For the purposes of this chapter linear algebra package (LAPACK) and basic linear algebra subprograms (BLAS) methods implemented as part of the LINPACK benchmark are used both to measure performance and to demonstrate the error analysis capabilities of the co-processor system. The benchmark implements methods for determining the solution to a dense $n \times n$ system of linear equations $Ax = b$ using Gaussian Elimination with partial pivoting [43]. The implementation used for testing is a C port of the original Fortran implementation available from [163]. Performance testing and error analysis have

been performed using array sizes from $n = 10$ to $n = 200$ and the value of A and b set using the `matgen` method provided as part of the LINPACK implementation. Statistical measurements have been performed using the Euclidean, (L^2), norm of the result vector, $x[n]$, defined as:

$$||x|| := \sqrt{x_1^2 + ... + x_n^2} \tag{5.16}$$

In order to determine a performance measure for comparison purposes the LINPACK has been run on a desktop using two configurations with settings listed in Table 5.5. In the first instance the benchmark is executed as normal in order to determine the baseline FP performance of the test system, and secondly using a SW implementation of MCA, in this case Monte Carlo arithmetic library (MCALIB) was used as the example SW implementation. At this stage it is not possible to set the optimization level of the SW implementation of MCA beyond O0, (disabled). This is most likely due to an incompatibility with with source code generated with the C intermediate language (CIL) compiler and one or more of the optimization options enabled by moving to level O1 and beyond. For the purposes of testing the performance of standard acIEEE-754 FPU is set to O3, (maximum optimization), however it should be noted that comparison testing has shown that using this optimization level achieves an approximate increase in performance of 2×. Performance results for both single and double precision test cases are shown in Figure 5.4. The average FP performance of the test system measured without using MCA was approximately 2 GFLOPS, while the average performance when using MCA was approximately 1.5 MFLOPS. These results represents a decrease in measured performance of three orders of magnitude and highlights the disadvantages faced by SW-based implementations.

5.3.2 System Performance Results

In order to test the performance of the co-processor the LINPACK benchmark was modified in order to be compatible with the Zynq 7 PS and to enable offloading of

System Parameters	
CPU	Intel Core 2 Duo E8400 (3 GHz)
Memory	4 GB
operating system (OS)	Ubuntu 12.04 LTS (32-bit)
GNU compiler collection (GCC) Version	4.6.3
Compilation Flags (IEEE-754)	O3
Compilation Flags (MCA)	O0

Table 5.5: PC specifications for baseline performance measurements of an IEEE-754 FPU

FP operations to the co-processor. Modification to the program include converting the timing function `SECOND` to utilize the AXI timer/counter module as opposed to ctime methods, implementing a MCA library containing a set of functions that send FP arithmetic functions to the co-processor, and finally modifying the LINPACK source to ensure each relevant FP operation calls a library function instead of using standard C FP operators. The modifications to the LINPACK source were performed using the cilly compiler as used in MCALIB to ensure all FP operations were correctly transformed to library calls. Using the Xilinx software development kit (SDK) a board support package (BSP) is defined containing a library of required functions for the Zynq and associated peripherals. An AXI4-Lite peripheral is implemented with a memory mapped interface, using the Vivado tool chain a set of driver functions are defined as part of the Vivado HLS IP export process. Included in this set of driver functions are functions for read/write access to each input and output register as well as functions for initialization, starting and polling the state of the peripheral. For the purposes of performance testing with the LINPACK benchmark the modified version of the SW is now compiled using GCC under Xilinx SDK with the O3 optimization flag. Testing has been conducted for array sizes $N = 10$ to $N = 200$ and results of testing are presented in Figure 5.5. The average performance of the co-processor over the tested array sizes was approximately 0.4 MFLOPS, this result is compared against measurements of SW based MCA implementation run using a desktop PC, measured as approximately 1.5 MFLOPS over the same array size range. These results demonstrate that the HW implementation of MCA is significantly

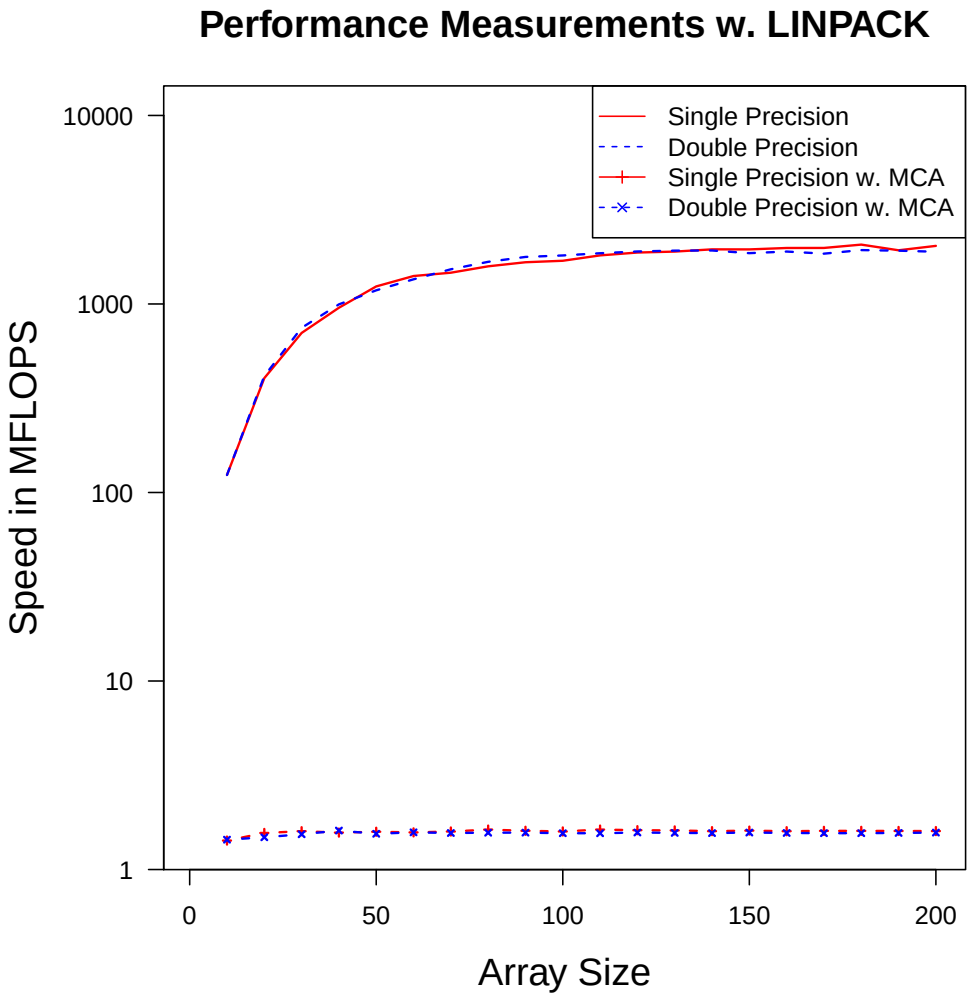


Figure 5.4: Comparison of performance of IEEE-754 FPU and MCA SW implementation measured with the LINPACK benchmark

outperformed by the SW implementation, and is in fact 75% slower than the original implementation. The lack of performance in the co-processor is due to two reasons;

- **Latency**: a high number of clock cycles occur between each operation, (and average of 24 cycles), can be addressed through pipeline.

- **Input/Output**: a significant portion of execution time is dedicated to transferring data from the processor to the co-processor, can be addressed using a high speed interface such as direct memory access (DMA).

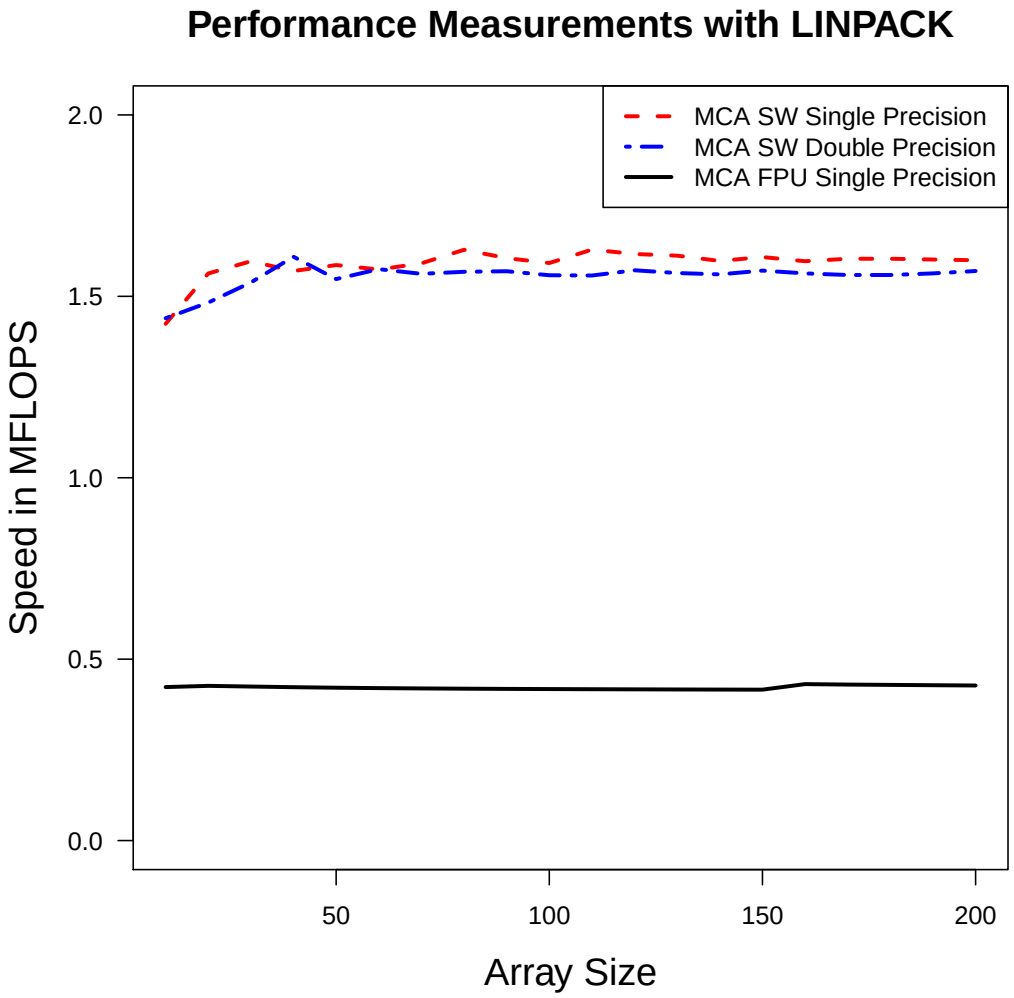


Figure 5.5: Comparison of performance of MCA FPU Co-Processor and MCA SW implementation measured with the LINPACK benchmark

Modifications to the co-processor to address these performance bottlenecks and updated performance results are presented in Section 5.4.

5.3.3 Results of Error Analysis

In addition to measuring the performance of the co-processor the L^2 norm of the result vector x has been measured using MCA error analysis techniques developed for Section 4.3 in order to verify the error analysis and detection capabilities of the co-processor. Testing has been conducted for array sizes between $N = 10$

and $N = 200$ with a step size of 10, using virtual precision values between $t = 1$ and $t = 24$ with a step size of 1. The number of trials for each experiment is set by the minimum required sample size of 100, (as detailed in Section 4.2.3), for a total of $20 \times 24 \times 100 = 48000$ executions of the benchmark. The results for error analysis as performed by the co-processor unit have been compared to the results for the same set of experiments as performed by MCALIB in order to ensure correct implementation of MCA on the co-processor. Detailed results for the $N = 100$ case are presented in Figures 5.6(a) and 5.6(b), where results for the co-processor are shown in figure 5.6(a) and results from MCALIB are shown in figure 5.6(b). Note that the values for both K and t_{min} are highly similar for both sets of results, with the difference being attributable to the large number of non-normal data points in this result set.

(a) Results of analysis for $N = 100$ using MCA FPU

(b) Results of analysis for $N = 100$ using MCALIB

Figure 5.6: MCA analysis of LINPACK benchmark

5.4 Customization of Co-Processor Implementation

As stated in the previous section the FP performance of the MCA FPU is significantly worse than that of a comparable SW implementation. In order to increase the performance of the co-processor to a level comparable with the SW implementation the performance bottlenecks in the system must be addressed. This has been achieved by customizing the co-processor to take advantage of vectorization operations in the LINPACK benchmark. By targeting these types of operations with a custom implementation the co-processor may be pipelined in order to reduce the initiation interval between operations, and implement a streaming DMA interface to reduce communications overhead between the processor and co-processor.

5.4.1 Profiling Results for LINPACK Benchmark

In order to customize the co-processor implementation to the LINPACK benchmark the SW must first be profiled in order to determine where the majority of execution time is spent and if these areas can be targeted for optimization strategies. For the purposes of this work the LINPACK benchmark has been profiled using the call graph execution profiler GNU profiler (GPROF) [69]. Profiling has been performed using the testing platform detailed in Table 5.5 with optimization level O0, the default profiler sampling period of 10 ms and over array sizes between $N = 10$ and $N = 1000$. Full results of profiling for all array sizes and methods are presented in Figure 5.7 and a summary of the mean and maximum execution times for each method is presented in Table 5.6. The results of profiling clearly show that a significant majority of execution time is spent in the functions `daxpy_r` and `daxpy_ur`. These two functions both implement the BLAS routine `daxpy` [136] which performs a double precision multiply accumulate (MAC) operation using two vectors each containing a total of N elements, X and Y, and one scalar quantity a:

$$Y = Y + aX \tag{5.17}$$

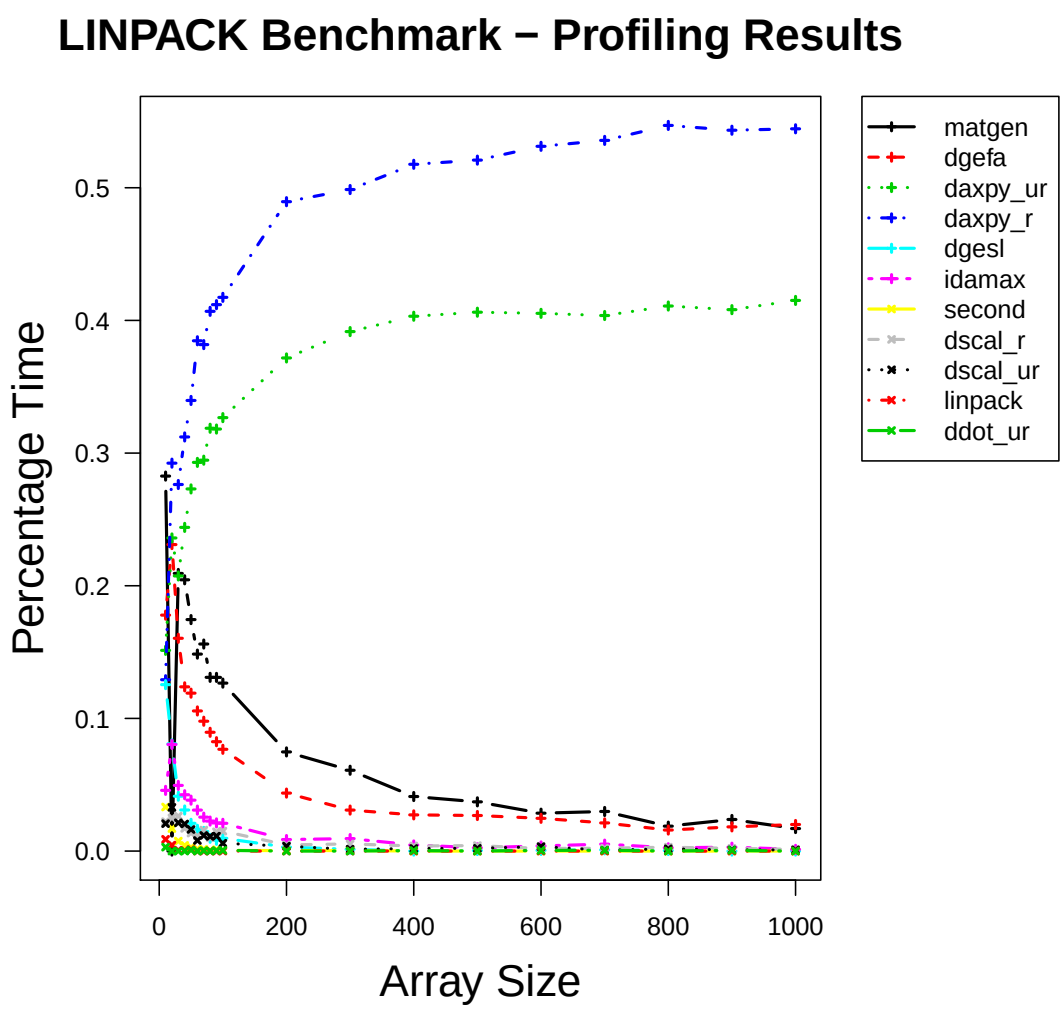


Figure 5.7: Results of profiling the LINPACK benchmark

Percentage Execution Time Results			
Name	**Type**	**Mean % Execution Time**	**Max % Execution Time**
`daxpy_r`	BLAS	42.53%	54.70%
`daxpy_ur`	BLAS	33.04%	41.51%
`matgen`	*overhead*	*9.98%*	*28.27%*
`dgefa`	LAPACK	7.86%	23.10%
`idamax`	BLAS	2.20%	8.03%
`dgesl`	LAPACK	1.92%	12.55%
`dscal_r`	BLAS	1.05%	2.63%
`dscal_ur`	BLAS	0.93%	3.29%
`second`	*overhead*	*0.37%*	*3.32%*
`linpack`	*overhead*	*0.08%*	*0.89%*
`ddot_ur`	BLAS	0.04%	0.30%

Table 5.6: Profiling Results - Average & maximum execution time over array sizes from $N = 10 \times 10$ to $N = 1000 \times 1000$

The BLAS implementation of the function utilizes loop unrolling in order to maximize performance, however in the event that the stride of the vectors X or Y is not equal to 8 bytes loop unrolling is not used. In order to provide a more complete performance measure the LINPACK benchmark provides both rolled and unrolled implementations of the function, named `daxpy_r` and `daxpy_ur`. For the purposes of this work the arithmetic operation of both functions is identical and both functions may be vectorized using the MCA FPU co-processor in the same way. As such the profiling results of both functions may be combined and it is evident that approximately 75% and up to 95% of execution time is spent performing a `daxpy` operation. Based on the profiling results of the benchmark it is possible to estimate the possible performance increase available using Amdahl's Law [5]:

$$R = \frac{1}{(1-P) + \frac{P}{S}} \tag{5.18}$$

where R is the potential performance improvement factor, P is the proportion of execution time to be sped up and S is the speed-up factor. Based on the estimated minimum speed of the original co-processor of 2.8 MFLOPS, the benchmark performance measurement of 0.45 MFLOPS and assuming a maximum possible pipelined

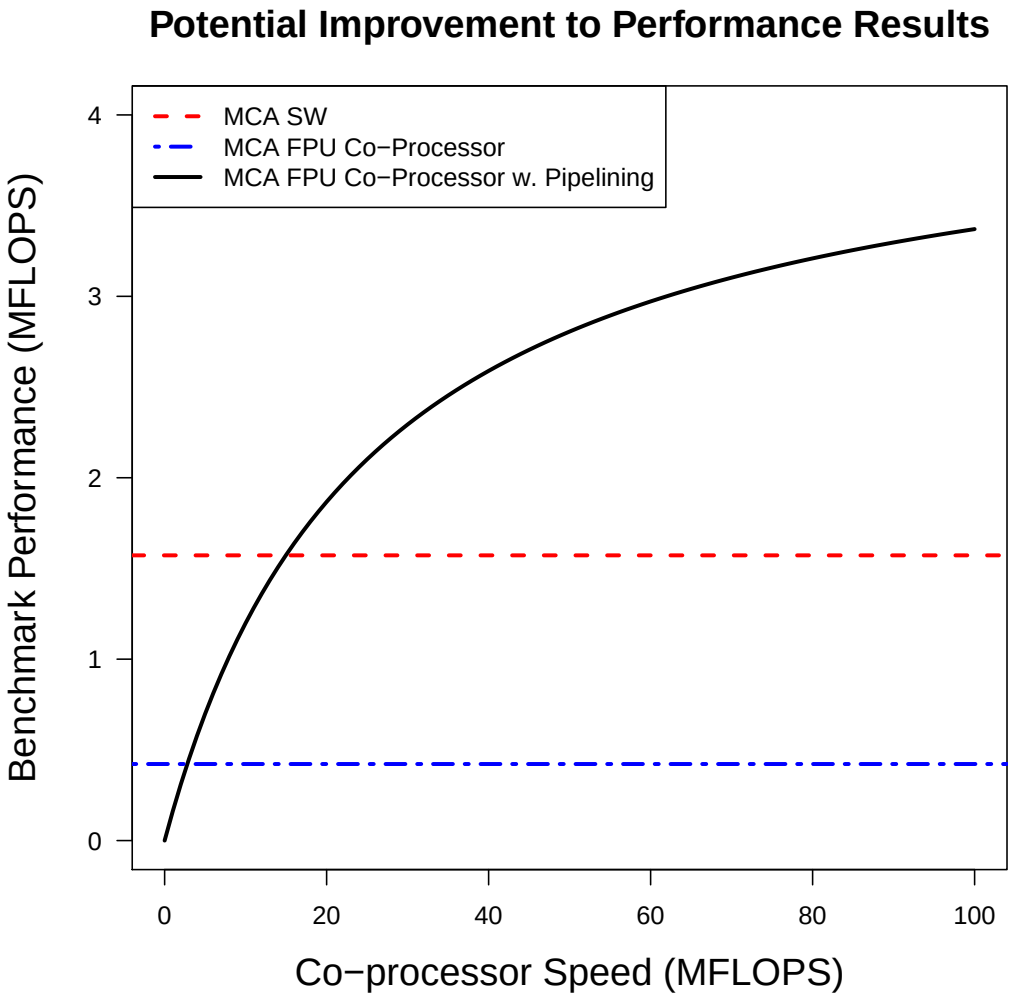


Figure 5.8: Potential improvement to benchmark performance based on profiling results

performance of the co-processor implementation is 100 MFLOPS, (an estimate based on the timing results from synthesis and a minimum initiation interval of 1 clock cycle), a speed-up factor of 6.5× to 8.5× is achievable depending on IO overhead. A complete estimate of the potential benchmark performance is shown in Figure 5.8 and compared against the performance measurement of the original co-processor implementation and the MCA SW implementation.

5.4.2 Modifications to Co-Processor & Bus Implementations

Having determined the functions for which optimization will provide the most benefit, the co-processor can now be modified in order to create a custom implementation that targets these types of operations. In order to maintain the original functionality of the co-processor, (i.e maintain the general purpose functionality in addition to the new optimized functions), the co-processor will now contain two sub-cores. One core consists of the original co-processor, accessed via an AXI4-lite interface and performing the original five arithmetic operations, while the second core contains a pipelined MAC operation with a streaming DMA interface capable of performing vectorized `daxpy` operations. Using Vivado HLS the required modifications to the original co-processor are minimal. The original MCA operations are modified to perform the MAC operation in terms of the inexact function:

$$y_i = \text{round}(\text{inexact}(\text{inexact}(y_i) + \text{inexact}(\text{inexact}(a) \times \text{inexact}(x_i)))) \tag{5.19}$$

Pipelining is implemented by adding the pipelining directive to the project with the interval option set to one clock cycle. The interface is modified using the interface directive with the AXI stream option enabled for the vector inputs and output to the co-processor. A total of three stream interfaces are implemented for the inputs X and Y, and the result vector. The 32-bit config input used to set the value of t, and the 64-bit scalar quantity a are implemented using AXI4-lite interfaces. This has been done to reduce the communications overhead, as the values a and t only need to be transferred to the co-processor once for every N operations, (where N is the length of the vectors X and Y), and therefore do not require a stream interface. Once these changes have been made the new co-processor core may be synthesized and exported as an IP Catalogue using the settings listed in Table 5.1. Preliminary synthesis results from HLS are shown in Table 5.7. Note that these results are for the pipelined co-processor core only and complete resource utilization estimates can be found by totalling the results from Tables 5.2 and 5.7. Having implemented

Timing Results (Estimated)		
Max Frequency	116 MHz	
Clock Uncertainty	1.25 ns	
Latency	21	
Pipeline Initiation Interval	1	
Resource Utilization (Estimated)		
BRAM	0 / 280	0 %
DSP	14 / 220	6 %
FF	7657 / 106400	7 %
LUT	15482 / 53200	29 %

Table 5.7: Vivado HLS Synthesis Results - Vectorized MCA FPU Co-Processor

the necessary changes to the co-processor the Zynq PS and associated interface modules must also be updated in order to correctly implement the processor/co-processor interface. In addition to the AXI master interface port, a high performance AXI4-lite slave port, `S_AXI_HP0` is enabled on the PS. This port is connected to a second AXI interconnect module, implemented with three AXI4-lite slave ports, each connected to a AXI4-lite master port on a DMA module, and one AXI4-lite master port, connected to the AXI4-lite slave port on the PS. The original AXI interconnect is also modified to include a further three AXI4-lite master ports, two of which are connected to the slave ports on a DMA module and the third to the slave port on the new co-processor core. As the pipelined co-processor core is implemented with two streaming inputs and one streaming output, the DMA interface is implemented using a pair of LogiCORE IP AXI DMA 7.1 modules [1], one implemented with both read and write channels, and one implemented with a write channel only. Each DMA unit is setup with a memory map data width of 64-bits, a stream data width of 64-bits and a maximum burst size of 256. The scatter/gather engine on both units is disabled. A summary of the post-implementation resource utilization report is shown in Table 5.8. Note that the table presents results of logic utilization for the pipelined co-processor core only and does not include resources used by the original co-processor or peripheral overhead such as the interconnect, timer, reset or DMA modules.

Resource Utilization			
Type	**Used**	**Available**	**Utilization**
Slice LUT	6241	53200	11.5%
Slice Registers	7214	106400	7%
F7 Multiplexers	13	26600	$<1\%$
F8 Multiplexers	1	13300	$<1\%$
Slices	2578	13300	19.5%
LUT as Logic	6241	53200	11.5%
LUT as Memory	0	53200	0%
LUT/FF Pairs	8421	53200	16%
BRAM	0	140	0%
DSP	14	220	6.5%

Table 5.8: Vivado Implementation Results - MCA FPU Co-Processor Pipelined Implementation

5.4.3 System Performance Results

The final step in modifying the co-processor was re-running the LINPACK benchmark in order to evaluate the performance benefits of the modifications. Minor modifications to the benchmark and to the MCA library used in the previous tests were required. These included implementing a function to set the values of the scalar quantities using the AXI4-lite interface, and a `daxpy` function to send and receive the vector quantities using the DMA interface. This function also included operations to flush/invalidate the cache before sending and after receiving data in order to maintain cache coherency. The LINPACK benchmark was also modified so that the MAC operations in the `daxpy_r` and `daxpy_ur` functions were offloaded to the co-processor. Initial testing of the co-processor was performed to determine the raw performance of the `daxpy` core when performing pipelined operations via the DMA interface. This testing was performed without the LINPACK benchmark and instead the time taken to perform a set of 1000 MAC operations was measured and used to determine the FLOPS of the pipelined core. These measurements result showed an average speed of approximately 25 MFLOPS, a significant improvement over both the original co-processor and over the SW implementation of MCA. Comparing these measurements against the predicted performance results detailed in Figure 5.8 indicates a benchmark performance of 2 to 3 MFLOPS is to be expected. Testing

with the LINPACK benchmark has been performed using the same system and compilation parameters as used for testing of the original co-processor implementation, and using array sizes between $N = 10$ and $N = 200$. Results of testing are shown in Figure 5.9 and are compared against the performance results of the MCA SW implementation. The peak performance as measured with the benchmark for an array size of $N = 200$ was approximately 3 MFLOPS, twice the speed of the SW implementation and 7.5$\times$ faster than the original co-processor implementation.

Several conclusions may be drawn by comparing the theoretical maximum throughput of the pipelined implementation of 100 MFLOPS, the tested maximum throughput of 25 MFLOPS, and the tested result of the LINPACK benchmark of 3 MFLOPS. Firstly IO overhead from the DMA interface results in a four-fold decrease in performance, i.e. an average of 3 clock cycles of processor to co-processor communication (and vice-versa) are required for every 1 clock cycle of computation. Secondly the final performance result is limited not only by the throughput of the co-processor but by the proportion of operations that can be optimized, the value of P in Amdahl's law as presented in equation 5.18. Improvements to the performance of the complete system may therefore be made in one of three ways, by decreasing the IO overhead, by improving the ratio of optimizable operations, (both of which may be achieved by selecting larger problem sizes), or by implementing several co-processor units in parallel to perform N independent MCA trials simultaneously. Given N parallel units operating on problems where $P \rightarrow 1$ throughputs of several hundred MFLOPS are achievable, leading to performance improvements of at least 1-2 orders of magnitude over existing SW implementations.

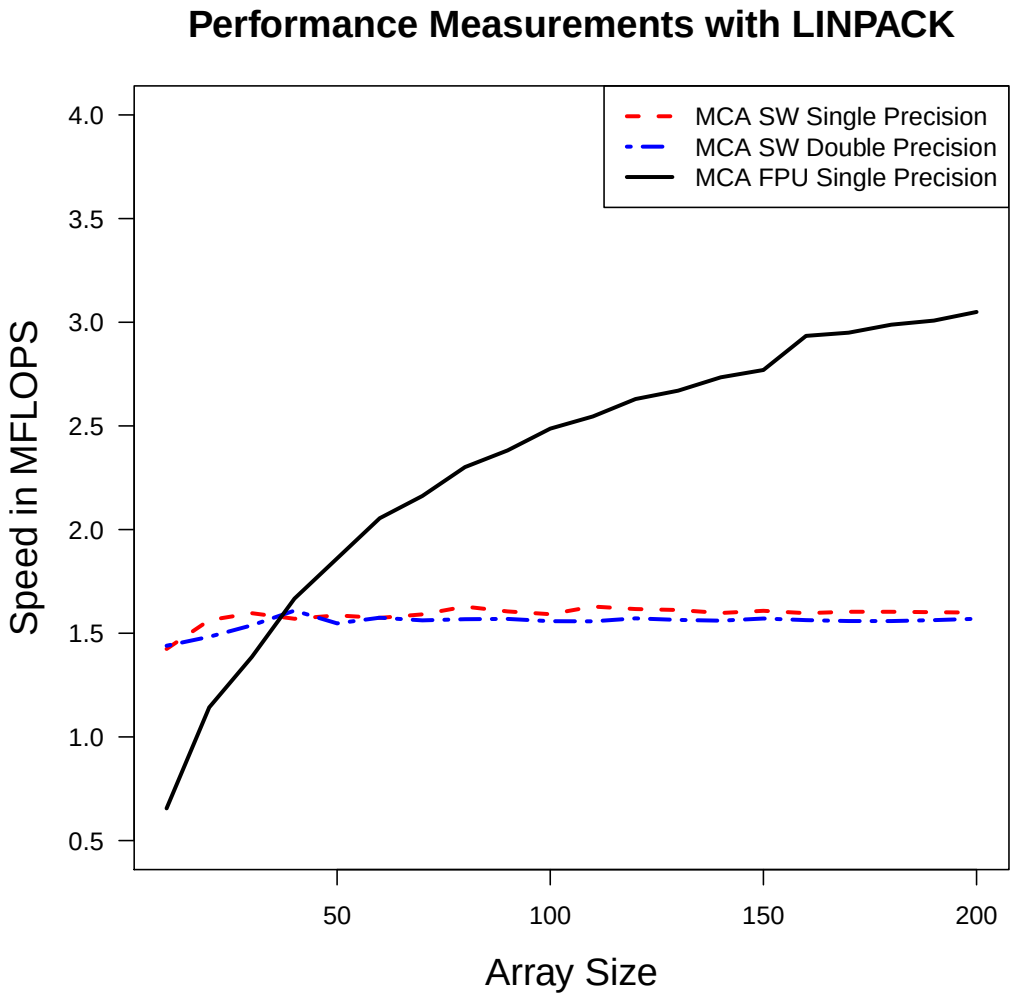


Figure 5.9: Comparison of performance of pipelined MCA FPU Co-Processor and MCA SW implementation measured with the LINPACK benchmark

5.5 Summary

Using high level design synthesis a complete system for run-time detection of round-off error has been devised. The design has been implemented as a co-processor to demonstrate the effectiveness of hardware acceleration of error detection algorithms. Testing of the co-processor system in combination with analysis methods for MCA developed for Chapter 4 have verified the implementation of MCA operators. Initial performance testing of the co-processor implementation has demonstrated that co-processor throughput and communications overhead are significant performance

bottlenecks and have resulted in an implementation that is in fact 75% slower than an equivalent SW implementation. In order to address these performance issues HLS design techniques have been used in order to develop a customized co-processor that targets vectorizable operations in the system undergoing analysis. Further testing of this optimized solution has shown that the performance of the current implementation is similar to an equivalent PC based SW implementation. This work shows that HW accelerated implementations of error detection algorithms can provide accurate measurements of the effects of rounding error while not impacting device performance.

Chapter 6

Variance Reduction Methods for Monte Carlo Arithmetic

6.1 Introduction

One of the drawbacks of Monte Carlo arithmetic (MCA) is the significant degradation in performance due to two primary causes. Firstly, probabilistic analysis necessitates N independent trials to be performed in order to provide a representative sample set for statistical analysis. While the independent nature of the trials means that they may be implemented with a high degree of parallelism, an N-fold increase in computation is required nevertheless. In the second case performance is degraded due to the additional computational overhead of the analysis affecting floating point (FP) performance. In the case of MCA the computation of the inexact function requires the generation and scaling of the random perturbation values ξ for input and output operands. In addition, conversion from a FP operation to an MCA operation requires the precision of the operators be extended to the MCA working precision, defined in Chapter 4, Section 4.2.2 as $W = p + t$. This can be achieved in one of three ways. If both single and double precision analysis are to be supported the existing FP hardware (HW) can be modified, (as modern floating point unit (FPU)s are typically double precision units). While this is achievable using

field programmable logic (FPL), a more feasible solution is to use a software (SW) library supporting extended precision FP. Finally the extended precision W may be simulated by restricting analysis to single precision FP types and implementing MCA operations using double precision FP, thus avoiding the requirement to modify existing FP HW or switch to a SW implementation.

In this thesis both SW based and HW accelerated implementations of MCA have been explored. In the case of Monte Carlo arithmetic library (MCALIB) as presented in Chapter 4 the extended precision is handled using Multiple Precision Floating-Point Reliably (MPFR) and this move from a FPU to a SW based implementation results in a $1300\times$ decrease in performance. When the number of minimum required number of trials $N = 100$ is also taken into account it is apparent that implementation of MCA will increase execution time by five orders of magnitude. In Chapter 5 a HW accelerated MCA FPU was presented demonstrating the feasibility of field programmable gate array (FPGA) based implementations of MCA. Using this setup the performance achieved was comparable to an equivalent SW implementation running on a desktop PC. However, while the raw performance of the optimized co-processor core was capable of a potential $25\times$ speed-up over the SW implementation under ideal conditions, once the repeated trials are taken into account this still represents a performance decrease of 3 orders of magnitude compared to standard FP performance on the same machine. Further performance gains are achievable via improvements to the MCA algorithm by reducing the required number of trials.

The concept of variance reduction techniques, designed to reduce the required number of trials by improving the convergence rate of an estimator, is well understood as it pertains to standard Monte Carlo method (MCM)s [46, 76, 98]. However, the application of these methods to MCA has yet to be explored. This chapter presents results of implementation and testing of two variance reduction techniques for MCA, the remainder of the Chapter is organized as follows. In section 6.2 im-

plementation of the variance reduction methods and the required modifications to MCA are presented. The experimental procedure for testing and comparing the results of implementation to standard MCA is presented in Section 6.3. Results of analysis for several test cases are presented in Section 6.4 and finally a summary is presented in Section 6.5.

6.2 Application of Variance Reduction Methods to MCA

6.2.1 Quasi-Monte Carlo Arithmetic

Quasi-Monte Carlo methods, as introduced in Section 3.3, are an implementation of the MCM where randomized inputs to the simulation are generated using an s-dimension low discrepancy sequence (LDS) as opposed to a uniformly distributed random number sequence. The LDS exhibits better equi-distribution of random variables for lower values of N, thus improving the convergence rate of a Monte Carlo estimator. The use of quasi-Monte Carlo techniques and LDSs is well understood as it pertains to standard MCMs [7, 45, 130, 156] and it has been shown quasi-Monte Carlo methods have an approximation error limited by $\mathrm{O}\left(\frac{\log N^s}{N}\right)$, while traditional MCMs have a probabilistic error of $\mathrm{O}\left(\frac{1}{\sqrt{N}}\right)$. Several studies have compared the convergence rate of quasi-Monte Carlo methods with that of MCMs when applied to practical real world problems, in particular it is noted that both MCMs and quasi-Monte Carlo methods converge to an accurate result quickly when applied to multi-dimensional integrals with a large number of dimensions, (300 and higher) [130, 156], however it has also been noted that in order for there to be a significant advantage over traditional MCMs, the number of dimensions, s should be small and the number of samples, N, should be large [112, 130]. Further studies have compared different types of LDS [113, 130] to determine if the convergence rate is dependent on sequence type. In particular the Sobol [159], Faure [49], Neiderreiter [135] and Halton [74] sequences have been compared, with the Sobol sequence commonly found to provide better convergence rates, particularly for problems with

higher dimensions, (more than 6).

For implementing quasi-Monte Carlo arithmetic (qMCA), the inexact function is re-defined as the *quasi*-exact function:

$$\text{qexact}(x, t, \xi_k) = x + \beta^{e_x - t}\xi_k \tag{6.1}$$

where $x \in \mathbb{F}$ is an FP number as defined in Section 2.4, t is the virtual precision of the qMCA operation and ξ_k is a random variable in the range $[-\frac{1}{2}, \frac{1}{2})$ drawn from the k^{th} dimension of the s-dimension Sobol sequence, (where $k \leq s$). Using this new definition an operation $\circ \in \{+, -, \times, \div\}$ is now implemented in terms of the qexact function:

$$x \circ y = \text{round}(\text{qexact}(\text{qexact}(x) \circ \text{qexact}(y))) \tag{6.2}$$

The dimensionality of the operation is determined by the number of unique variables present, i.e. the operation $a = b + c$ contains three unique variables and as such requires a Sobol sequence with three dimensions:

$$a = \text{round}(\text{qexact}(\text{qexact}(b, t, \xi_1) + \text{qexact}(c, t, \xi_2), t, \xi_3)) \tag{6.3}$$

Alternatively an operation such as $a = a + b$ contains only two unique variables and can be implemented using a two-dimension Sobol sequence:

$$a = \text{round}(\text{qexact}(\text{qexact}(b, t, \xi_1) + \text{qexact}(c, t, \xi_2), t, \xi_1)) \tag{6.4}$$

where the random number ξ_1 is drawn from the first dimension twice. Extending this concept to a complete FP algorithm and/or program is detailed in Section 6.2.3. The implementation of the Sobol sequence used for qMCA has been selected for its ability to generate high-dimension Sobol sequences, as a typical FP program

will contain several thousand independent variables and will therefore require a Sobol sequence with several thousand dimensions for analysis. The Sobol sequence generator that has been selected is capable of generating a sequence with number of dimensions $s = 21201$ [90, 91].

6.2.2 Monte Carlo Arithmetic with Antithetic Variates

As in the case of quasi-Monte Carlo Methods, correlated sampling with antithetic variates is a variance reduction technique that has been applied to standard MCMs [75, 77]. As detailed in Section 3.2.3 implementing this sampling type requires modifying the input to the simulation such that the s-dimension uniform random number sequence of length N is replaced with a uniform random number sequence of length $\frac{N}{2}$ combined with its antithetic path. In practice, this will mean that a single iteration of a Monte Carlo simulation is now performed with two simulation paths, which are subsequently combined to determine the estimated value, as shown in equations 3.30 through 3.33. When the results of Monte Carlo simulation from the original and antithetic paths are combined the variance in the result is given by:

$$\sigma^2(z) = \frac{1}{4}\left[\sigma^2(z_1) + \sigma^2(z_2) + 2\,\mathrm{cov}(z_1, z_2)\right] \tag{6.5}$$

as shown in equation 3.34. In the event that z_1 and z_2 are independent and identically distributed (IID) the resulting variance of z simplifies to the following:

$$\sigma^2(z) = \frac{\sigma^2(z_1)}{2} \tag{6.6}$$

$$= \frac{\sigma^2(z_2)}{2} \tag{6.7}$$

as $\sigma^2(z_1) = \sigma^2(z_2)$ and the covariance $\mathrm{cov}(z_1, z_2)$ self-cancels [20]. The use of antithetic variates for correlated sampling ensures that z_1 and z_2 are no longer IID and the covariance is negative, reducing the variance in the final result.

Implementation of Monte Carlo arithmetic with antithetic variates (ATMCA) requires redefinition of the inexact function to form the *antithetic*-exact function:

$$\text{atexact}(x, t, \xi_k) = [x - \beta^{t-e_x}\xi_k, x + \beta^{e_x-t}\xi_k] \tag{6.8}$$

where $x \in \mathbb{F}$ and t are defined as in the inexact, (equation 3.50), and qexact functions, (equation 6.1), and ξ_k is a random variable in the range $[-\frac{1}{2}, \frac{1}{2})$ drawn form the k^{th} dimension of the s-dimension uniform random sequence, (where $k \leq s$). Using this redefinition, the results of applying the atexact function to a variable x results in an interval containing two results:

$$\text{atexact}(x, t, \xi_k) = [x_1, x_2] \tag{6.9}$$

where x_1 is the result of randomization of x using the random variable ξ_k and x_2 the result of randomization using the antithetic path $-\xi_k$. Using this definition of the atexact function a FP operation $z = x \circ y \rightarrow \circ \in \{+, -, \times, \div\}$ is implemented as follows:

$$[z_1, z_2] = \text{round}(\text{atexact}(\text{atexact}(x) \circ \text{atexact}(y))) \tag{6.10}$$

$$z = \frac{z_1 + z_2}{2} \tag{6.11}$$

In practice it is simpler to modify the random number generator to alternate between generated values and the antithetic path to form a random number stream $\xi = [\xi_1, -\xi_1, \xi_2, -\xi_2, ..., \xi_{\frac{N}{2}}, -\xi_{\frac{N}{2}}]$ for each dimension, as opposed to tracking results that are formed as a set of interval bounds. This method will correctly implement ATMCA if the following conditions are met:

- The dimensionality of the problem is tracked correctly, (see Section 6.2.3).
- The number of trials, N, is even.
- The random number generator is not reset or reseeded between trials.

6.2.3 Determining Problem Dimension with Data Flow

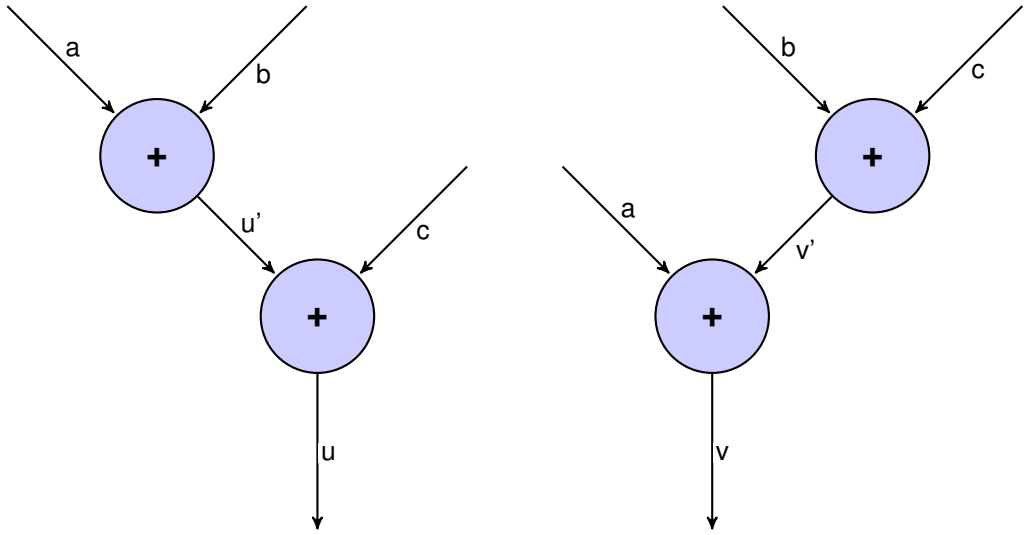
As noted in the descriptions of qMCA and ATMCA in the preceding sections the implementation of these techniques necessitates that the dimensions of the problems under analysis be carefully tracked. In the case of MCA the separate dimensions of the s-dimension uniformly distributed random number set may be drawn from the same psuedo-random number generator (PRNG) without issue, and as such there is no requirement to track the separate dimensions. When implementing qMCA and ATMCA the nature of the number sequences requires that individual dimensions be tracked in order to correctly apply the qexact and atexact functions. In the case of qMCA each problem dimension must be assigned a separate dimension of the Sobol sequence in order to ensure equi-distribution of the random number inputs in that dimension, without which the benefits of quasi-Monte Carlo, (fast convergence of the result estimators), will not be realized. In the case of ATMCA a random number vector, $[x_1, x_2, ..., x_{\frac{N}{2}}]$, and its antithetic path, $[-x_1, -x_2, ..., -x_{\frac{N}{2}}]$ must be combined and applied to a single dimension in order to correctly implement correlated sampling.

The number of dimensions in a FP algorithm or SW implementation is determined by the number of independent variables present. This means that the dimensionality of the problem may be tracked via data flow analysis and represented visually by a data flow graph (DFG), where nodes represent qMCA or ATMCA operations and edges represent separate dimensions. This concept is demonstrated for the two results, u and v, of Knuth's example:

$$u = (a + b) + c \tag{6.12}$$

$$v = a + (b + c) \tag{6.13}$$

in Figures 6.1(a) and 6.1(b) respectively. Note that in each DFG there are two nodes and five edges, as such qMCA or ATMCA analysis is implemented for Knuth's



(a) DFG of variable u in Knuth's example (b) DFG of variable v in Knuth's example

Figure 6.1: Data flow analysis of Knuth's example

example with two operations, (or six instances of the qexact or atexact functions), and five dimensions. For example, qMCA analysis of u is implemented as follows:

$$u' = \text{round}(\text{qexact}(\text{qexact}(a, t, \xi_1) + \text{qexact}(b, t, \xi_2), t, \xi_3)) \tag{6.14}$$

$$u = \text{round}(\text{qexact}(\text{qexact}(u', t, \xi_3) + \text{qexact}(c, t, \xi_4), t, \xi_5)) \tag{6.15}$$

Note that the variable u' is common to both operations, and therefore is represented by a single edge on the DFG in Figure 6.1(a) and is randomized by a common input to the qexact function, ξ_3. The resulting qMCA implementation will require a 5-dimension Sobol sequence, where dimensions 1, 2, 4 and 5 are polled once per trial, and dimension 3 is polled twice per trial.

6.2.4 Limitations of Implementation & Scalability

At present qMCA and ATMCA have been implemented using a modified form of MCALIB. The modifications performed allow for problem dimension to be tracked for small scale examples by re-implementing the interface functions to include

information on the address of incoming variables and the return address of calling functions. Each variable is identified and stored in a look up table (LUT) with the following identifiers:

- Variable address
- Return address of the current function.
- Return address of the caller of the current function.

While MCALIB has been modified the **cilly** compiler has not, and as such implementation of qMCA and ATMCA requires manual modification of existing source code. Furthermore the current system for monitoring dimension only checks the return address two steps up the function call stack. A complete and automated implementation would therefore require the following modifications (at minimum);

- Monitoring of problem dimension should include variable address and n levels of recursion in the function call stack.
- The **cilly** compiler must be modified to correctly re-write FP operations as function calls to the modified version of MCALIB
- If no disable option is available, the C intermediate language (CIL) must be modified to prevent replacement of array indexing with fixed point pointer arithmetic, as this removes information on variable address.

In addition to these limitations the size of the problem that can be analysed using qMCA is also limited. As stated in Section 6.2.1 the implementation of the Sobol sequence selected for this work has a dimension limit of $s = 21201$. While this is the highest dimensionality for a C-based generator available at the time of writing, this s value still represents a strict limitation on the dimensionality of the problem space under analysis. For example, experiments conducted using the linear equations software package (LINPACK) benchmark in the previous sections were performed using array sizes up to 200 by 200, a problem that would required a random number

sequence with at least $s = 40000$ dimensions. This limit would seem to restrict the use of quasi-Monte Carlo method (qMCM) to problems with lower dimensionality, a restriction further enforced by the previously made point that the benefits of qMCM over standard MCM are diminished as the number of dimensions becomes large.

At this stage modifications to CIL and the development of extremely large dimension quasi-random number generators are outside the scope of this thesis and are left for future implementation. Instead the variance reduction extensions to MCA are presented as theory with proof of concept examples provided.

6.3 Analysis & Comparison of Implementation Types

Analysis of the proposed techniques has been performed by comparing the experimental results of qMCA and ATMCA against MCA for a set of three test cases:

- Knuth's example:

$$u = (111111113 - 11111111) + 7.5111111 \tag{6.16}$$

$$v = 11111113 + (-11111111 + 7.5111111) \tag{6.17}$$

- FP summation:

$$s = \sum_{i=1}^{n} x_i, \text{ for } n \geq 3 \tag{6.18}$$

- The Chebyshev polynomial

$$T_{20}(z) = \cos(20\cos^{-1}(z)) \tag{6.19}$$

$$\begin{aligned} &= 524288z^{20} - 2621440z^{18} + 5570560z^{16} \\ &- 6553600z^{14} + 4659200z^{12} - 2050048z^{10} \\ &+ 549120z^{8} - 84480z^{6} + 6600z^{4} \\ &- 200z^{2} + 1 \end{aligned} \tag{6.20}$$

These comparisons have been made in order to verify the improved rate of convergence resulting from the application of variance reduction techniques, specifically reducing the approximation error in the results from $\mathrm{O}\left(\frac{1}{\sqrt{N}}\right)$ to $\mathrm{O}\left(\frac{1}{N}\right)$. In this chapter the experimental procedure is outlined, and methods used for measurement and comparison of result convergence rates are detailed. A modified version of MCALIB has been used for the implementation of qMCA and ATMCA analysis, and the standard version of MCALIB has been used to generate MCA results for comparison.

6.3.1 Experimental Procedure

In the general case MCA analysis of a function $y = f(x)$ implemented as FP SW is tested by performing repeated executions. For the purposes of this chapter a **trial** is defined as a single execution of the function $f(x)$. Using MCALIB standard experimental procedure as defined in Chapter 4 is to sweep the virtual precision between t_{start} and t_{end}, typically set to 1 and 53 respectively for analysis of double precision FP, and to perform a total of N trials for each t value. The minimum value for N has been defined as 100 trials. For the purposes of this chapter an **experiment** is defined as a set of $t_{end} \times N$ trials. In order to determine the effect of sample size on the results the value of N is swept from 5 to $N_{max} = 200$. This results in a set of 195 experiments producing 195 data sets each containing $t_{end} \times N$ data points. Collected data is analyzed using statistical measurements as detailed in Sections 6.3.2

through 6.3.3.

6.3.2 Sample Mean

Using standard MCMs the true mean of the result is determined by measuring the sample mean, as such the sample mean is designated an estimator of the true mean. The sample mean is measured by first grouping data by sample size and virtual precision. Grouping data in this way creates $195 \times 53 = 10335$ subsets, each containing N samples, and the sample mean and variance are calculated as follows:

$$\overline{\mu}_{N,t} = \frac{1}{N} \sum_{i=5}^{N} y_i \tag{6.21}$$

where y is the result of the function $y = f(x)$ under testing. The approximation error of an estimator is measured by calculating the mean squared error (MSE) over N. This is measured for each value of N as follows:

$$\text{MSE}_{N,t}(\hat{\theta}_{N,t}) = E[(\hat{\theta}_{N,t} - \theta)^2] \tag{6.22}$$

where $\text{MSE}_{N,t}(\hat{\theta}_{N,t})$ is the MSE of the estimator for a specific value of sample size, N, and virtual precision, t, $\hat{\theta}_{N,t}$ is the measured result of the estimator, and θ is the true result. Using this setup a set of N MSE values is determined for each t value by calculating the following:

$$\text{MSE}_{N,t}(\overline{\mu}_{N,t}) = \frac{1}{N} \sum_{i=5}^{N} (\overline{\mu}_{i,t} - \mu)^2 \tag{6.23}$$

In the case of sample mean, each result is compared against the true result of the function under test, represented by the true mean, μ.

As is the case with standard MCM, the approximation error as estimated with the MSE should decrease with N in a monomial relationship, (i.e. a linear relationship

when plotted on a log-log scale), modelled as:

$$\log_{10}(y) = m \log_{10}(x) + c \tag{6.24}$$

$$y = 10^c \cdot x^m \tag{6.25}$$

where $y = \text{MSE}_{N,t}(\hat{\theta}_{N,t})$ and $x = N$. Linear regression analysis with log transformed variables is performed using $\text{MSE}_{N,t}(\hat{\theta}_{N,t})$ as the dependent variable and N as the exploratory variable to determine the value of m, which represents the convergence rate of the estimator.

6.3.3 Stopping Criteria

Stopping criteria or stopping rules are intended to provide methods to automatically terminate a Monte Carlo simulation once the required level of accuracy in the result estimator has been achieved. Stopping rules for standard MCMs are often implemented using analysis of the confidence intervals on the sample mean [62]. Alternatively, stopping rules as used in statistical analysis will typically define application specific rules or equalities that must be met for a specified number of iterations. For the purposes of testing in this chapter a stopping rule has been defined based on MCA analysis methods developed as part of MCALIB and presented in Chapter 4. This stopping rule compares sequential values of significant figures lost, K, and minimum required precision, t_{min} calculated over a range of sample sizes N in order to find the minimum sample size $n \geq m$ for which the following inequalities hold:

$$\delta_K > \max_{i=n+1}^{n+m} [K_i - K_{i-1}] \tag{6.26}$$

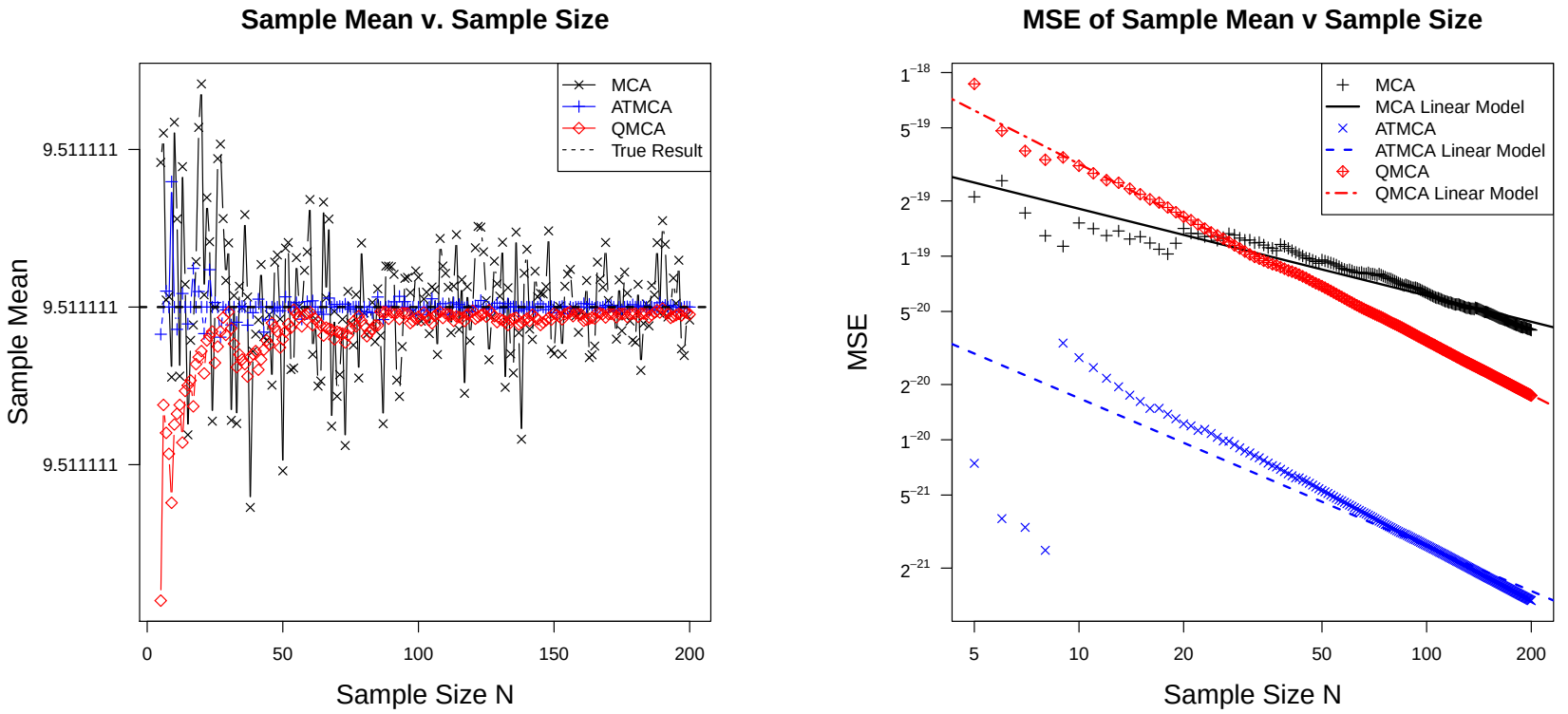
$$\delta_t > \max_{i=n+1}^{n+m} [t_i - t_{i-1}] \tag{6.27}$$

i.e. the inequalities:

$$\delta_K > K_n - K_{n-1} \tag{6.28}$$

$$\delta_t > t_n - t_{n-1} \tag{6.29}$$

should hold for m sequential iterations.



(a) Sample mean

(b) MSE of sample mean

Figure 6.2: Analysis of sample mean of Knuth's example for $t = 53$ and sample sizes from $N = 5$ to $N = 200$

Method Type	Sample Mean (Slope)	Sample Std. Dev. (Slope)
MCA	-0.566	0.153
ATMCA	-0.899	0.167
qMCA	-0.970	0.009

Table 6.1: Summary of linear regression analysis of sample mean results for virtual precision values from $t = 1$ to $t = 53$ for Knuth's example

6.4 Results

6.4.1 Analysis of Estimator Convergence

In the first case the sample mean and the MSE of the sample mean have been measured for Knuth's example, and the $t = 53$ case is plotted in Figures 6.2(a) and 6.2(b). This figure clearly shows the estimators from all three methods converging towards the true result. However it is clear that in the case of qMCA and ATMCA the sample mean is converging to the true result significantly faster than in the case of MCA. Performing the linear regression analysis for all three sets of results produces in the following models, (for the $t = 53$ case only):

$$\text{MSE}_{mca} = 10^{-18.27} \cdot N^{-0.47} \tag{6.30}$$

$$\text{MSE}_{qmca} = 10^{-17.53} \cdot N^{-0.97} \tag{6.31}$$

$$\text{MSE}_{atmca} = 10^{-18.97} \cdot N^{-0.81} \tag{6.32}$$

Note that the error in the MCA estimator is proportional to $\text{O}\left(\frac{1}{\sqrt{N}}\right)$ while the error in the qMCA case is proportional to $\text{O}\left(\frac{1}{N}\right)$. While the sample mean of the ATMCA results converges at a faster rate than that of the MCA results, it is not proportional to $\text{O}\left(\frac{1}{N}\right)$. The three models are compared against the raw data in Figure 6.2(b). Statistical measurements on the results of linear regression analysis on all virtual precision values from $t = 1$ to $t = 53$ are summarized in Table 6.1. These measurements confirm the general case for Knuth's sample demonstrating the effectiveness of qMCA, in this case the variance reduction method results in a significant improvement in the convergence of the result estimator over the standard MCA method.

Method Type	Sample Mean (Slope)	Sample Std. Dev. (Slope)
MCA	-0.561	0.228
ATMCA	-0.416	0.216
qMCA	-0.909	0.328

Table 6.2: Summary of linear regression analysis of sample mean results for virtual precision values from $t = 1$ to $t = 53$ for floating point summation

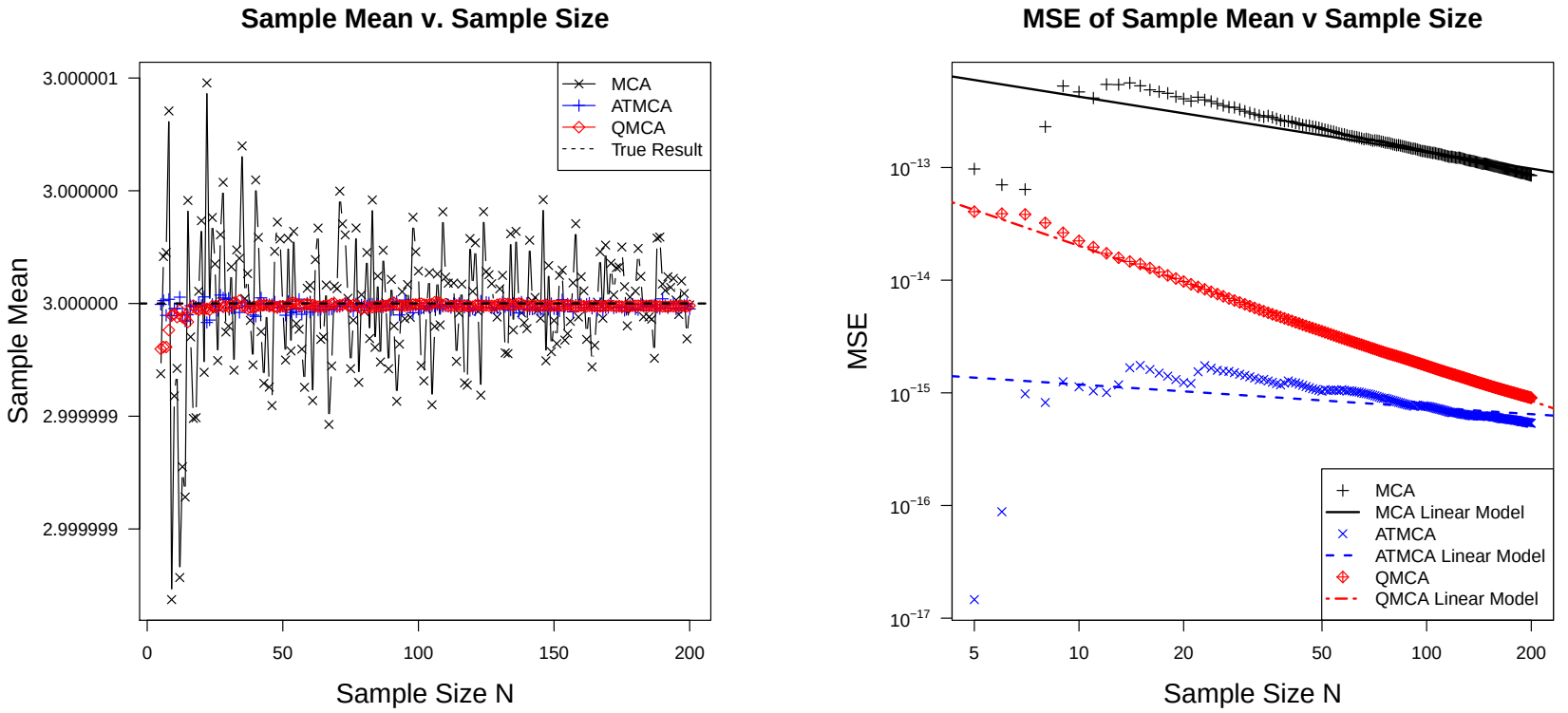
This same type of analysis has been performed for results of MCA analysis of the FP summation example. Results of measuring the sample mean and the MSE of the sample mean for the $t = 24$ case are shown in Figures 6.3(a) and 6.3(b) respectively. As was the case of Knuth's example, the sample mean of the results of qMCA analysis is clearly converging towards the true result at a faster rate than that of either MCA or ATMCA analysis. This finding is re-enforced by the result of the linear regression analysis, highlighted in Figure 6.3(b) for the $t = 24$ case. The complete results for each model are as follows:

$$\text{MSE}_{mca} = 10^{-12.00} \cdot N^{-0.43} \tag{6.33}$$

$$\text{MSE}_{qmca} = 10^{-12.57} \cdot N^{-0.95} \tag{6.34}$$

$$\text{MSE}_{atmca} = 10^{-13.37} \cdot N^{-0.75} \tag{6.35}$$

Note again the difference in convergence rates between MCA and qMCA, ($\text{O}\left(\frac{1}{\sqrt{N}}\right)$ vs. $\text{O}\left(\frac{1}{N}\right)$). Results of statistical measurements of the linear regression analysis results for all cases from $t = 1$ to $t = 53$ are summarized in Table 6.2.



(a) Sample mean

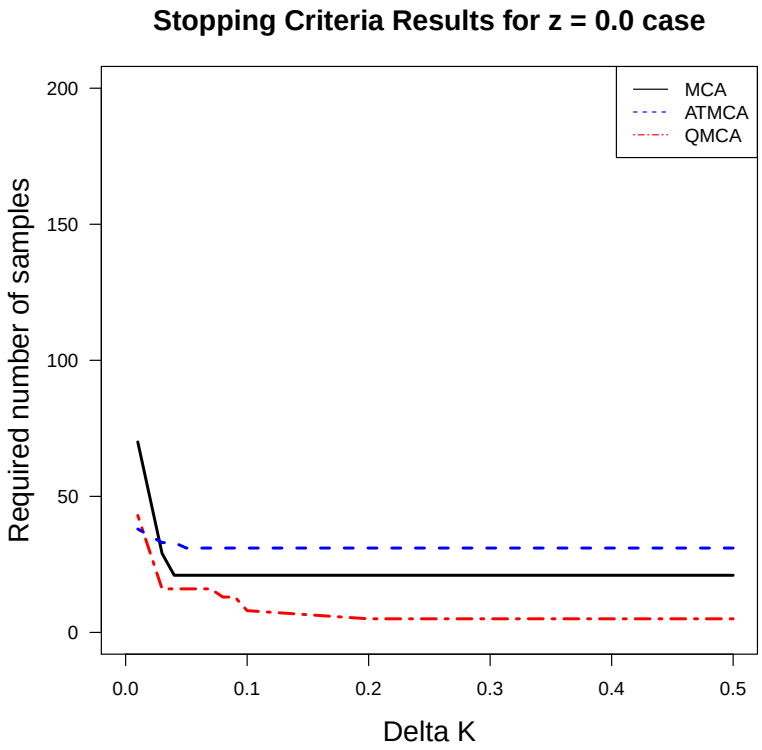
(b) MSE of sample mean

Figure 6.3: Analysis of sample mean of summation example for $t = 24$ and sample sizes from $N = 5$ to $N = 200$

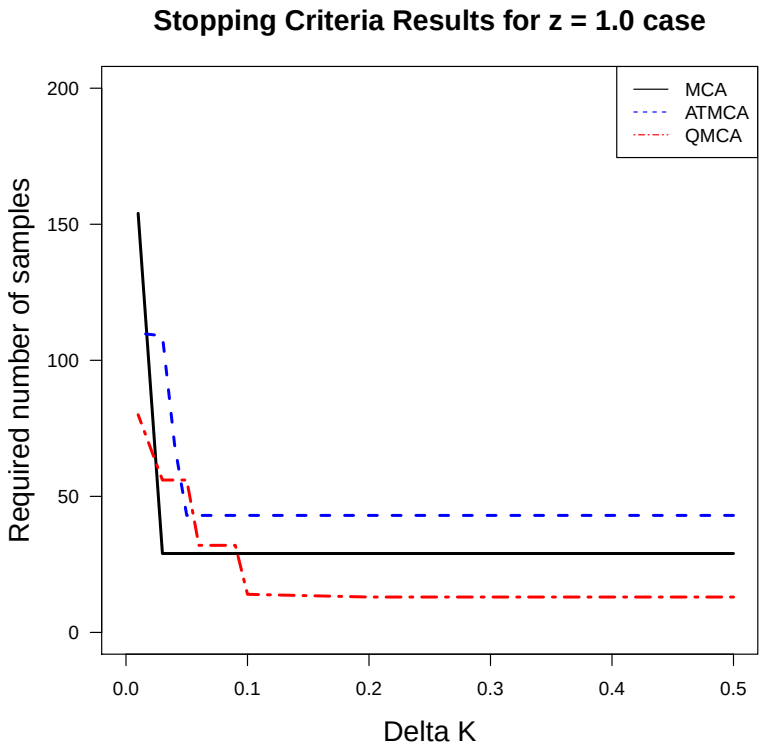
6.4.2 Analysis of Stopping Criteria

The final set of experiments conducted for this chapter have been performed using the Chebyshev polynomial as a test case, and have been conducted using stopping rules to demonstrate the reduction in sample size requirements resulting from the use of quasi-Monte Carlo techniques. The testing procedure has been expanded to test multiple values of the Chebyshev input z, (as in Chapter 4), in this case testing the known best and worst case inputs, $z = 0.0$ and $z = 1.0$. The first set of results are shown for both input test cases in Figures 6.4(a) and 6.4(b). In these results the value of the stopping criteria threshold δ_k has been varied in order to determine its effect on the results of analysis and on required sample size. In the analysis method developed as part of MCALIB, the bound for the result of K when used to determine outliers was set at ± 0.5, i.e. plus or minus half a binary digit and as such the value of δ_K has been varied between 0.5 and 0.0025 for testing purposes while the value of m is fixed at 3. The results indicate that the stopping criteria stabilizes at $\delta_K = 0.2$ and show that the required sample size for qMCA is significantly lower than that of MCA in this region. For the $z = 0.0$ case the average result in the stable region for qMCA is 5 samples and 21 samples for MCA, a reduction of approximately 75%. In addition $\sqrt{21} = 4.58...$ or approximately 5 samples. For the $z = 1.0$ case the average result is 13 samples for qMCA and 29 samples for MCA, a reduction of approximately 55%. Note that in this case the result for qMCA is significantly higher than $\sqrt{29} = 5.39...$ indicating the ideal reduction in variance has not been achieved.

Figures 6.5(a) and 6.5(b) detail the same set of results as the previous figure, however the experiment has been modified to sweep the value of m while fixing the value of $\delta_k = 0.25$. From these results two findings are apparent. The results of stopping criteria analysis are highly data dependent, indicating that more samples are required when analysing systems exhibiting high sensitivity to rounding error. Secondly it is again clear that the results for qMCA analysis require a smaller number of samples than that of MCA or ATMCA.

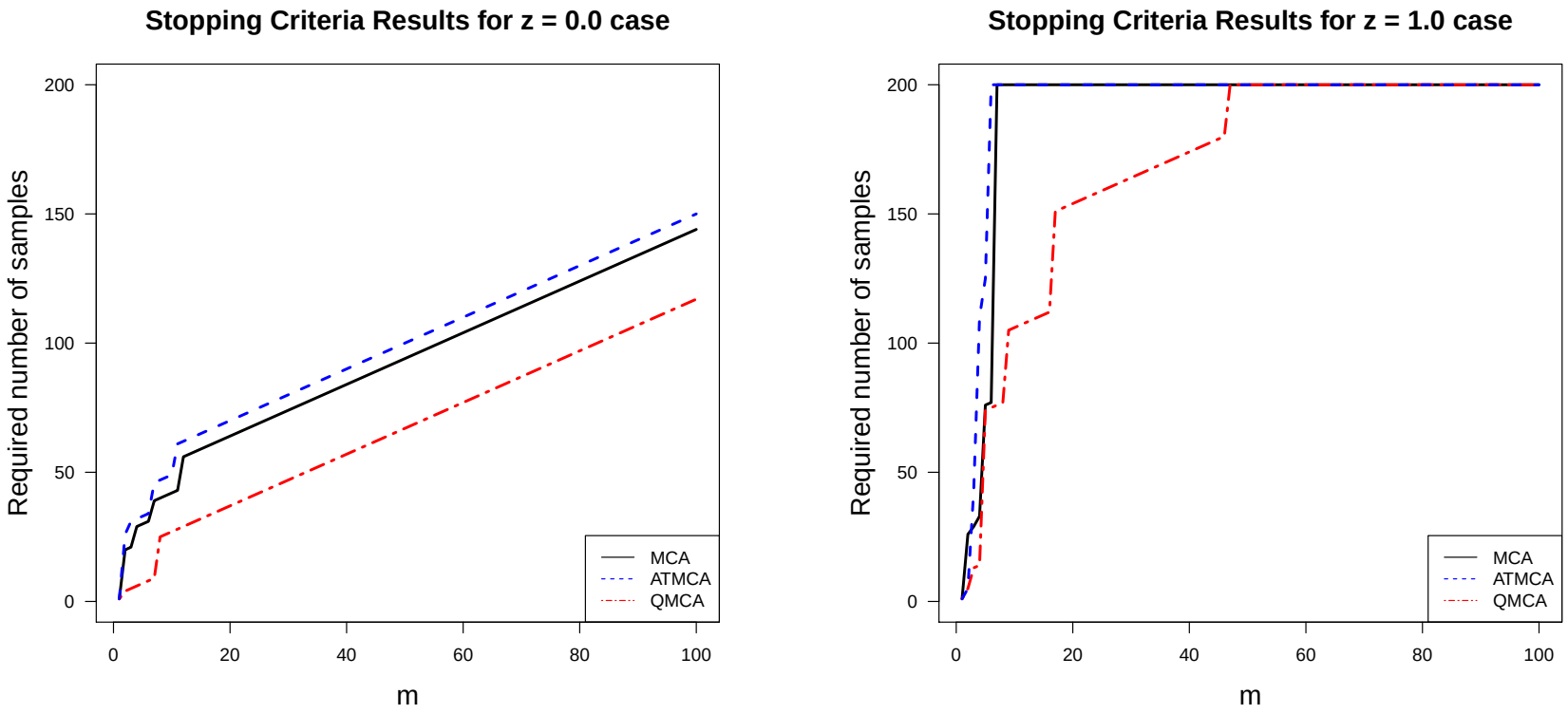


(a) $z = 0.0$ case



(b) $z = 1.0$ case

Figure 6.4: Analysis of required number of samples compared with δ_K for Chebyshev polynomial



(a) $z = 0.0$ case

(b) $z = 1.0$ case

Figure 6.5: Analysis of required number of samples compared with m for Chebyshev polynomial

6.5 Summary

In this chapter the application of variance reduction techniques to MCA has been investigated with the aim of reducing the required number of samples for effective analysis. Two methods have been implemented, qMCA using an LDS in place of the uniform random number generator using in standard MCA, and ATMCA, and implementation of correlated sampling using antithetic variates. Testing for several test cases has demonstrated that the qMCA analysis type significantly improves the convergence rate of the measured estimator, sample mean, and reduces the required sample size under certain conditions. The results of testing the ATMCA method are not as promising, while the convergence rate of the sample mean was found to increase in some test cases, the increase was not as significant as that of qMCA, and did not appear in as many cases. Furthermore analysis of stopping criteria testing indicated that ATMCA increases the required sample size when the stopping criteria is based on measurements of K and t_{min}. As these measurements are calculated from the relative standard deviation (RSD) of results, it is possible that ATMCA implementation has an undesirable effect on the sample variance of the result set.

This work has shown that certain variance reduction techniques as applied to traditional MCMs are also effective when applied to MCA and can achieve $\mathrm{O}\left(\frac{1}{N}\right)$ convergence compared with $\mathrm{O}\left(\frac{1}{\sqrt{N}}\right)$ for the naive case. However the reduction in sample size comes at the cost of increased computational complexity. In the case of qMCA and ATMCA the dimension of the problems under analysis must be tracked in order to ensure correct results, this increases the computational time required for analysis when compared with MCA. As with standard MCMs these techniques are best applied in cases where the number of dimensions is low, reducing the computational overhead and providing the best chance for increasing the convergence rate of the results.

Chapter 7

Conclusion

7.1 Introduction

The field of numeric analysis has grown as the use of computer arithmetic systems has become more widespread, leading to the development of static and dynamic error analysis methods. However, the amount of knowledge required of both computer arithmetic and numeric analysis has limited the use of error analysis as part of the software development life cycle (SDLC). This lack of understanding beyond experts in the field stems from issues with implementation, performance and scalability. This thesis is intended to address these issues by developing automated tools for the implementation of Monte Carlo arithmetic (MCA).

7.2 Findings & Contributions

The specific aims and contributions of this thesis are listed in chapter 1, and experimental findings are detailed in chapters 4 through 6. The primary aim of this thesis was the development of methods that allow developers to understand rounding error and required precision level and the key contribution of this work lies in the proposed plots which allow a complete MCA analysis to be understood from an easily interpretable summary. These plots are generated from the results of auto-

mated implementations of MCA; the Monte Carlo arithmetic library (MCALIB) and the MCA floating point unit (FPU). These systems have been designed to allow the discovery of optimized implementations by both experts in numeric analysis and lay persons. A simple approach may be implemented by automatically translating all floating point (FP) operations to MCA operations. Alternatively, a more in-depth analysis may be achieved by focusing on specific operations or by selecting the type of random perturbations applied to the operations. MCA can be applied using precision bounding, (applied to input operands only), or by random rounding, (applied to output operands). By selecting either precision bounding or random rounding a user may compare results and determine to what level rounding error or catastrophic cancellation is affecting the stability of the system under analysis, thus providing a method for more detailed analysis. This work was also addressed issues with performance and scalability, encouraging adoption by reducing the cost of implementation typically associated with these methods. The MCA FPU demonstrated a tested throughput of 2 to 25 MFLOPS depending on conditions, an improvement of up to 16× over equivalent software (SW) implementations, a figure that can be increased to over 100× if improvements are made to communication between processor and co-processor or with parallel implementations. Performance has been further improved by extending MCA to quasi-Monte Carlo arithmetic (qMCA). This has reduced the convergence rate of the estimation error from $\mathrm{O}\left(\frac{1}{N}\right)$ to $\mathrm{O}\left(\frac{1}{\sqrt{N}}\right)$ and in test cases the required number of samples was reduced by a factor of 2 to 5 times.

Encouraging the adoption of error analysis methods is a critically important goal, particularly as the use of computer arithmetic systems becomes increasingly more widespread. As the number of computing devices and applications for these devices grows, increased understanding of error and the effects of error will be necessary. MCALIB has been provided as an open source implementation and it is hoped that this will lead to further developments and improvements to MCA and other dynamic error analysis tools. Wider adoption of error analysis routines

will also have the obvious effect of mitigating the effects of rounding error. The current understanding of error is that significant effects are few and far between, and therefore can often be discounted except in safety critical systems. This thinking is contradicted by two points,

1. The number of computing devices and their performance is increasing rapidly, therefore increasing the likelihood of errors occurring.

2. Discounting the possibility of errors as negligible can increase the effects of these errors when they do occur.

This second point is especially important to consider, as the cost of catastrophic errors and failures can often be heavy, as in the case of Intel's floating point division (FDIV) bug, the failure of the Patriot air defense system at Dharan air base or the failure of the Ariane 5 rocket system [65, 105, 140]. Wider implementation of error analysis schemes and more importantly, better understanding of the concepts of error has the potential to not only limit the incidence rate of error, but to mitigate its effects through well designed arithmetic systems and SW. Development of systems with the effects of error in mind also has the potential to improve efficiency and reduce overhead. In the case of optimization or iterative refinement problems, minimizing the sensitivity to rounding error will allow for faster convergence rates to be achieved, reducing the level of computation required and the cost of implementation.

7.3 Future Work

The automated application of other dynamic error analysis techniques could be investigated with the aim of developing a library containing a set of dynamic error analysis techniques including the Contrôle et Estemation Stochastique des Aarondis Calculs (CESTAC) method, interval arithmetic (IA), random interval arithmetic (RIA) and affine arithmetic (AA), creating a framework for dynamic error analysis. In addition this library could contain an implementation of qMCA allowing for

automated implementation of this method and investigation of the application of variance reduction techniques on larger scale problems. In the case of the MCA FPU, future work should investigate the implementation of custom FP libraries and integration with compiler systems capable of fully utilizing a pipelined co-processor, with the aim of improving performance and ease of use. Further hardware (HW) acceleration techniques should also be investigated, including graphics processing unit (GPU) based implementations or high performance computing (HPC) and large scale cluster systems for highly parallel implementations. The processor/co-processor type interface can be extended beyond the Xilinx Zynq platform to desktop based systems communicating with the field programmable gate array (FPGA) over peripheral component interconnect express (PCIe) interfaces as this approach will reduce transfer overheads. Finally this study has been limited to dynamic error analysis, combining it with static techniques would allow the advantages of both to be enjoyed.

List of Publications

[1] Michael Frechtling and Philip H. W. Leong. MCALIB - a tool for automated rounding error analysis. *ACM Transactions on Programming Languages and Systems*, 2014. In press, accepted 23 Aug 2014. (not cited)

[2] Michael Frechtling and Philip H. W. Leong. An fpga-based floating point unit for rounding error analysis. In Wayne Luk and George Constantinides, editors, *Transforming Reconfigurable Systems*. Imperial College Press, 2015. (not cited)

Bibliography

[1] Logicore ip axi dma v7.1. http://xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf. (cited page 135)

[2] J H Ahrens, U Dieter, and A Grube. Pseudo-random Numbers. *Computing*, 6(1-2):121–138, March 1970. (cited pages 64 and 65)

[3] JH Ahrens, U Dieter, and A Grube. Pseudo-Random Numbers - New Proposal for Choice of Multiplicators. *Computing*, 6:121–&, 1970. (cited page 67)

[4] A. Amaricai, M. Vladutiu, and O. Boncalo. Design of Floating Point Units for Interval Arithmetic. In *Research in Microelectronics and Electronics, 2009. PRIME 2009. Ph.D.*, pages 12 –15, july 2009. (cited page 52)

[5] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967. (cited page 132)

[6] MVA Andrade and JLD Comba. Affine Arithmetic. 1994. (cited page 49)

[7] Søren Asmussen and Peter W Glynn. *Stochastic Simulation: Algorithms and Analysis: Algorithms and Analysis*, volume 57. Springer, 2007. (cited pages 68 and 143)

[8] J. Asserrhine, J.M. Chesneaux, and J.L. Lamotte. Estimation of Round-Off Errors

on Several Computer Architectures. *Journal of Universal Computer Science*, 1(7):455–468, 1995. (cited page 51)

[9] C.R Baugh and B Wooley. A Two's Complement Parallel Array Multiplication Algorithm. *Computers, IEEE Transactions on*, (12):1045–1047, 1973. (cited page 10)

[10] OJ Bedrij. Carry-select adder. *Electronic Computers, IRE Transactions on*, (3):340–346, 1962. (cited page 9)

[11] P. Behrooz. Computer Arithmetic: Algorithms and Hardware Designs. *Oxford University Press*, 19, 2000. (cited pages 8 and 9)

[12] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and Implementation of a Special-Purpose Static Program Analyzer For Safety-Critical Real-Time Embedded Software. *The Essence of Computation*, pages 85–108, 2002. (cited page 45)

[13] Paul Bratley and Bennett L Fox. Algorithm 659: Implementing sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):88–100, 1988. (cited page 67)

[14] Richard P Brent. *Algorithms for Minimization without Derivatives*. Courier Dover Publications, 1973. (cited page 91)

[15] H. Brönnimann, G. Melquiond, and S. Pion. The Design of the Boost Interval Arithmetic Library. *Theoretical Computer Science*, 351(1):111–118, 2006. (cited page 51)

[16] Ashley W Brown, Paul HJ Kelly, and Wayne Luk. Profiling Floating Point Value Ranges for Reconfigurable Implementation. In *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing*, pages 6–16, 2007. (cited page 53)

[17] Ashley W Brown, Paul HJ Kelly, and Wayne Luk. Profile-Directed Speculative

Optimization of Reconfigurable Floating Point Data Paths. In *Proceedings of the Workshop on Reconfigurable Computing at HiPEAC*, 2008. (cited page 53)

[18] John W Carr III. Error analysis in floating point arithmetic. *Communications of the ACM*, 2(5):10–15, 1959. (cited page 2)

[19] James Case. Interval Arithmetic and Analysis. *The College Mathematics Journal*, 30(2):106–111, March 1999. (cited pages 46 and 47)

[20] Russell CH Cheng. The use of antithetic variates in computer simulations. *Journal of the Operational Research Society*, pages 229–237, 1982. (cited page 145)

[21] R. Chotin and H. Mehrez. A Floating-Point Unit using Stochastic Arithmetic Compliant with the IEEE-754 Standard. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, volume 2, pages 603 – 606 vol.2, 2002. (cited page 52)

[22] C W Clenshaw and F W J Olver. Beyond Floating Point. *Journal of the ACM*, 31(2):319–328, March 1984. (cited page 19)

[23] Georges Louis Leclerc comte de Buffon. *Essai d'arithmetique morale*. 1777. (cited page 56)

[24] George A Constantinides. Perturbation Analysis for Word-Length Optimization. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 81–90. IEEE, 2003. (cited page 52)

[25] George A Constantinides. Word-Length Optimization for Differentiable Nonlinear Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(1):26–43, 2006. (cited page 52)

[26] George A Constantinides, Peter YK Cheung, and Wayne Luk. The Multiple Wordlength Paradigm. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 51–60. IEEE, 2001. (cited page 52)

[27] George A Constantinides, Peter YK Cheung, and Wayne Luk. Optimum Wordlength Allocation. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 219–228. IEEE, 2002. (cited page 52)

[28] George A Constantinides, Peter YK Cheung, and Wayne Luk. Wordlength Optimization for Linear Digital Signal Processing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(10):1432–1442, 2003. (cited page 52)

[29] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. Iterative Dataflow Analysis, Revisited. *Proceedings of the (PLDI'02)*, 2002. (cited page 45)

[30] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977. (cited page 45)

[31] Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979. (cited page 45)

[32] L Dadda. Some Schemes for Parallel Multipliers. *Computer arithmetic*, 1980. (cited page 10)

[33] I.L Dalal, D Stefan, and J Harwayne-Gidansky. Low Discrepancy Sequences for Monte Carlo Simulations on Reconfigurable Platforms. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pages 108–113, 2008. (cited pages 63 and 64)

[34] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software (TOMS)*, 37(1):2, 2010. (cited page 52)

[35] AC Davies and YT Fung. Interfacing a hardware multiplier to a general-purpose microprocessor. *Microprocessors*, 1(7):425–432, 1977. (cited page 10)

[36] LH De Figueiredo. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms*, 2004. (cited page 49)

[37] Keith DeHaven. Extensible processing platform ideal solution for a wide range of embedded systems. *Extensible Processing Platform Overview White Paper*, 2010. (cited page 114)

[38] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an Industrial use of FLUCTUAT on Safety-Critical Avionics Software. *Formal Methods for Industrial Critical Systems*, pages 53–69, 2009. (cited pages 3 and 45)

[39] David Delmas and Jean Souyris. Astrée: From Research to Industry. *Static Analysis*, pages 437–451, 2007. (cited pages 3 and 45)

[40] J.E. Dennis and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, volume 16. Society for Industrial Mathematics, 1987. (cited page 97)

[41] Alain Deutsch. Static Verification of Dynamic Properties. *PolySpace White Paper*, 2003. (cited page 45)

[42] U Dieter. Pseudo-Random Numbers - Exact Distribution of Pairs. *Mathematics of computation*, 25(116):855–&, 1971. (cited pages 64 and 67)

[43] J.J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. (cited pages 96, 108, and 123)

[44] Alexander Dreyer. *Interval analysis of analog circuits with component tolerances*. Shaker Aachen, 2005. (cited page 3)

[45] Moshe Dror, Pierre L'ecuyer, and Ferenc Szidarovszky. *Modeling Uncertainty: An Examination of Stochastic Theory, Methods, and Applications*, volume 46. Springer Science & Business Media, 2002. (cited page 143)

[46] William L Dunn and J Kenneth Shultis. *Exploring Monte Carlo Methods*. Elsevier, 2011. (cited pages 58, 60, 61, 63, and 142)

[47] C Dutang and P Savicky. randtoolbox: Generating and testing random numbers. *R package*, 2013. (cited page 67)

[48] C.F Fang, Tsuhan 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*, 2003. Literature Review - Section 2: Mathematical theory of error analysis. (cited pages 50 and 51)

[49] Henri Faure. Discrépance de suites associées à un système de numération (en dimension s). *Acta Arithmetica*, 41(4):337–351, 1982. (cited page 143)

[50] Floating-point Working Group. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, 1985. (cited page 26)

[51] Floating-point Working Group. IEEE Standard for Radix-Independent Floating-Point Arithmetic. *ANSI/IEEE Std 854-1987*, 1987. (cited page 26)

[52] Floating-point Working Group. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1 –58, 2008. (cited pages 26 and 36)

[53] GE Forsythe and MA Malcolm. Computer Methods for Mathematical Computations. John Wiley & Sons, 1977. (cited pages 19 and 23)

[54] Ian Foster and Stephen Taylor. A compiler approach to scalable concurrent-program design. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):577–604, 1994. (cited page 82)

[55] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):13, 2007. (cited page 83)

[56] John Fox. Robust Regression. *An R and S-Plus companion to applied regression*, 2002. (cited page 91)

[57] Michael Frechtling and Philip H. W. Leong. MCALIB - a tool for automated rounding error analysis. *ACM Transactions on Programming Languages and Systems*, 2014. In press, accepted 23 Aug 2014. (cited page 4)

[58] Michael Frechtling and Philip H. W. Leong. An fpga-based floating point unit for rounding error analysis. In Wayne Luk and George Constantinides, editors, *Transforming Reconfigurable Systems*. Imperial College Press, 2015. (cited page 4)

[59] A Abdul Gaffar, Oskar Mencer, Wayne Luk, Peter YK Cheung, and Nabeel Shirazi. Floating-Point Bitwidth Analysis via Automatic Differentiation. In *Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on*, pages 158–165. IEEE, 2002. (cited page 52)

[60] Altaf Abdul Gaffar, Wayne Luk, Peter YK Cheung, Nabeel Shirazi, and James Hwang. Automating Customisation of Floating-Point Designs. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 523–533. Springer, 2002. (cited page 53)

[61] Altaf Abdul Gaffar, Oskar Mencer, and Wayne Luk. Unifying Bit-Width Optimisation for Fixed-Point and Floating-Point Designs. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 79–88. IEEE, 2004. (cited page 53)

[62] Michael J Gilman. A brief survey of stopping rules in monte carlo simulations. In *Proceedings of the second conference on Applications of simulations*, pages 16–20. Winter Simulation Conference, 1968. (cited page 153)

[63] Paul Glasserman. *Monte Carlo methods in financial engineering*, volume 53. Springer, 2004. (cited page 3)

[64] D Goldberg. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Computing Surveys*, 1991. (cited pages 2, 19, 40, and 41)

[65] David Goldberg. Computer Arithmetic. *Computer Architecture: A Quantitative Approach, David Patterson and John L. Hennessy, Eds. Morgan Kaufmann, Los Altos, Calif., Appendix A*, 1990. (cited pages 1, 19, 21, 23, 25, 38, 39, 41, 89, and 165)

[66] Robert E Goldschmidt. *Applications of division by convergence*. PhD thesis, Massachusetts Institute of Technology, 1964. (cited page 18)

[67] F. Goualard. GAOL: Not Just Another Interval Arithmetic Library, 2006. (cited page 51)

[68] E. Goubault and S. Putot. Static Analysis of Numerical Algorithms. *Static Analysis*, pages 18–34, 2006. (cited page 45)

[69] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982. (cited page 130)

[70] Stef Graillat, Fabienne Jézéquel, Shiyue Wang, and Yuxiang Zhu. Stochastic arithmetic in multiprecision. *Mathematics in Computer Science*, 5(4):359–375, 2011. (cited page 51)

[71] Group, ARITH Research. Hardware Algorithms for Arithmetic Modules. http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html. (cited pages 7 and 8)

[72] Alain Guyot, Bertrand Hochet, and J-M Muller. A way to build efficient carry-skip adders. *Computers, IEEE Transactions on*, 100(10):1144–1152, 1987. (cited page 9)

[73] JH Halton. A Retrospective and Prospective Survey of Monte Carlo Method. *SIAM review*, 12(1):1–&, 1970. (cited page 63)

[74] John H Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12):701–702, 1964. (cited page 143)

[75] JM Hammersley and KW Morton. A new monte carlo technique: antithetic variates. In *Mathematical proceedings of the Cambridge philosophical society*, volume 52, pages 449–475. Cambridge Univ Press, 1956. (cited page 145)

[76] John Michael Hammersley and David Christopher Handscomb. *Monte carlo methods*, volume 1. Springer, 1964. (cited pages 88 and 142)

[77] John Michael Hammersley and JG Mauldon. General principles of antithetic variates. In *Mathematical proceedings of the Cambridge philosophical society*, volume 52, pages 476–481. Cambridge Univ Press, 1956. (cited page 145)

[78] Eldon Hansen. Interval Arithmetic in Matrix Computations, Part I. *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, 2(2):308–320, January 1965. (cited page 46)

[79] Eldon Hansen and Roberta Smith. Interval Arithmetic in Matrix Computations, Part II. *SIAM Journal on Numerical Analysis*, 4(1):1–9, March 1967. (cited page 46)

[80] D.M. Harris and S.L. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2007. (cited page 11)

[81] John D Hey, Tibor M Neugebauer, and Carmen M Pasca. Georges-louis leclerc de buffon's 'essays on moral arithmetic'. In *The Selten School of Behavioral Economics*, pages 245–282. Springer, 2010. (cited page 56)

[82] Nicholas J Higham. *Accuracy and Stability of Numerical Algorithms*. Number 48. Siam, 1996. (cited pages 39, 40, 41, 42, 75, and 89)

[83] N.J. Higham. The Accuracy of Floating Point Summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993. (cited page 95)

[84] Michael G Hilgers. Quasi-Monte Carlo Methods in Cash Flow Testing Simulations. In *WSC '00: Proceedings of the 32nd Conference on Winter Simulation*. Society for Computer Simulation International, December 2000. (cited page 64)

[85] Peter J Huber. Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1):73–101, 1964. (cited page 92)

[86] Thang Viet Huynh and M Mucke. Error Analysis and Precision Estimation for Floating-point Dot-products using Affine Arithmetic. In *Advanced Technologies for Communications (ATC), 2011 International Conference on*, pages 319–322, 2011. (cited pages 49 and 50)

[87] Xilinx Inc. Zynq-7000 All Programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Accessed: 2015-24-03. (cited pages xiii and 115)

[88] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *Proceedings of the 5th international conference on Supercomputing*, pages 244–251. ACM, 1991. (cited page 82)

[89] F. Jézéquel and J.M. Chesneaux. CADNA: A Library for Estimating Round-off Error Propagation. *Computer Physics Communications*, 178(12):933–955, 2008. (cited page 51)

[90] Stephen Joe and Frances Y Kuo. Sobol sequence generator, September. (cited page 145)

[91] Stephen Joe and Frances Y Kuo. Constructing sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing*, 30(5):2635–2654, 2008. (cited page 145)

[92] W. Kahan. Pracniques: Further Remarks on Reducing Truncation Errors. *Communications of the ACM*, 8(1):40, 1965. (cited page 95)

[93] W Kahan. A Survey of Error Analysis. *Information Processing: Proceedings of the IFIP Congress*, 2:1214–1239, 1971. (cited pages 2 and 41)

[94] W Kahan. On a Proposed Floating-point Standard. *ACM SIGNUM Newsletter*, 1979. (cited pages 24, 25, and 38)

[95] W Kahan. IEEE Standard 754 for Binary Floating-point Arithmetic. *Lecture Notes on the Status of IEEE*, 1996. (cited pages 19, 21, 22, 25, and 26)

[96] W Kahan. *How Futile are Mindless Assessments of Round-off in Floating Point Computation*, 2006. http://www.cs.berkeley.edu/∼wkahan/Mindless.pdf. (cited pages 3, 19, 43, and 47)

[97] William Kahan. The improbability of probabilistic error analyses for numerical computations. In *UCB Statistics Colloquium, evans hall edition*, page 20, 1996. (cited pages xiii, 75, and 77)

[98] Herman Kahn and Andy W Marshall. Methods of reducing sample size in monte carlo computations. *Journal of the Operations Research Society of America*, 1(5):263–278, 1953. (cited page 142)

[99] Kimon Karras and James Krica. *Designing Protocol Processing Systems with Vivado High Level Synthesis*. Xilinx Application Note 1209. August 2014. (cited page 120)

[100] Gary A Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973. (cited page 45)

[101] R. Klatte and G.F. Corliss. *C-XSC: A C++ Class Library for Extended Scientific Computing*. Springer-Verlag, 1993. (cited page 51)

[102] D.E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional, 1997. (cited page 12)

[103] Israel Koren. *Computer arithmetic algorithms*. Universities Press, 2002. (cited page 11)

[104] William K Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*. Prentice Hall PTR, 2005. (cited page 44)

[105] G Le Lann. An Analysis of the Ariane 5 Flight 501 Failure - A System Engineering Perspective. In *Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on*, pages 339–346, 1997. (cited pages 38 and 165)

[106] P L'Ecuyer. Maximally Equidistributed Combined Tausworthe Generators. *Mathematics of Computation*, 1996. (cited page 115)

[107] Piere L'Ecuyer and Christiane Lemieux. Quasi-Monte Carlo via Linear Shift-register Sequences. In *WSC '99: Proceedings of the 31st Conference on Winter Simulation: Simulation - A Bridge to the Future*. ACM, December 1999. (cited page 63)

[108] Pierre L'Ecuyer and Christiane Lemieux. Recent advances in randomized quasi-monte carlo methods. In *Modeling uncertainty*, pages 419–474. Springer, 2005. (cited page 70)

[109] D-U Lee, A Abdul Gaffar, Ray CC Cheung, Oskar Mencer, Wayne Luk, and George A Constantinides. Accuracy-Guaranteed Bit-Width Optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(10):1990–2000, 2006. (cited page 52)

[110] Dong-U Lee, Altaf Abdul Gaffar, Oskar Mencer, and Wayne Luk. MiniBit: Bit-Width Optimization via Affine Arithmetic. In *Proceedings of the 42nd annual Design Automation Conference*, pages 837–840. ACM, 2005. (cited page 52)

[111] Christiane Lemieux. Randomized quasi-monte carlo: a tool for improving the efficiency of simulations in finance. In *Simulation Conference, 2004. Proceedings of the 2004 Winter*, volume 2, pages 1565–1573. IEEE, 2004. (cited page 70)

[112] Christiane Lemieux. *Monte Carlo and Quasi-Monte Carlo Sampling*, volume 20. Springer, 2009. (cited pages 70 and 143)

[113] George Levy. An introduction to quasi-random numbers. *Numerical Algorithms Group Ltd., http://www. nag. co. uk/IndustryArticles/introduction_to_quasi_random_numbers. pdf (last accessed in April 10, 2012)*, 2002. (cited page 143)

[114] P. Linz. Accurate Floating-Point Summation. *Communications of the ACM*, 13(6):361–362, 1970. (cited page 95)

[115] D.C. Liu and J. Nocedal. On the Limited Memory BFGS Method for Large Scale Optimization. *Mathematical programming*, 45(1):503–528, 1989. (cited page 96)

[116] Haw-Jing Lo and D.V Anderson. A Hardware-efficient Implementation of the Fast Affine Projection Algorithm. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 581–584, 2009. (cited page 50)

[117] Chris Lomont. Fast inverse square root. *Tech-315 nical Report*, 2003. (cited page 32)

[118] Harold M Lucal. Arithmetic Operations for Digital Computers Using a Modified Reflected Binary Code. *IEEE Transactions on Electronic Computers*, EC-8(4):449–458, December 1959. (cited pages 6 and 7)

[119] M.A. Malcolm. On Accurate Floating-Point Summation. *Communications of the ACM*, 14(11):731–736, 1971. (cited page 95)

[120] Peter Markstein. Software division and square root using goldschmidt's algorithms. In *Proceedings of the 6th Conference on Real Numbers and Computers (RNC'6)*, volume 123, pages 146–157, 2004. (cited page 32)

[121] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM*

Transactions on Modeling and Computer Simulation (TOMACS), 8(1):3–30, 1998. (cited page 67)

[122] N Metropolis. The beginning ofthe monte carlo method. *Los Alamos Science*, (15):125, 1987. (cited page 55)

[123] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 2004. (cited page 55)

[124] Nicholas Metropolis and Stanislaw Ulam. The monte carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949. (cited page 55)

[125] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12, 2008. (cited page 44)

[126] David Monniaux. The Pitfalls of Verifying Floating-point Computations. *ACM Transactions on Programming Languages and Systems*, 30(3):1–41, May 2008. (cited page 19)

[127] RE Moore. Interval Analysis, 1966. (cited pages 45 and 46)

[128] J.J. Moré and D.J. Thuente. Line Search Algorithms with Guaranteed Sufficient Decrease. *ACM Transactions on Mathematical Software (TOMS)*, 20(3):286–307, 1994. (cited page 97)

[129] Hozumi Morohosi and Masanori Fushimi. A practical approach to the error estimation of quasi-monte carlo integration. *Monte Carlo and Quasi-Monte Carlo Methods*, 377:390, 1998. (cited page 70)

[130] William J Morokoff and Russel E Caflisch. Quasi-monte carlo integration. *Journal of computational physics*, 122(2):218–230, 1995. (cited page 143)

[131] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, and Damien Stehle. *Handbook of Floating-Point Arithmetic*. November 2009. (cited pages 11, 19, 20, 21, 22, 23, 25, 39, and 52)

[132] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction*, pages 209–265. Springer, 2002. (cited page 82)

[133] Thi Viet Nga Nguyen and François Irigoin. Efficient and effective array bound checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):527–570, 2005. (cited page 82)

[134] H Niederreiter. Quasi-Monte Carlo Methods and Pseudo-random Numbers. *American Mathematical Society*, 1978. (cited page 63)

[135] Harald Niederreiter and NSF-CBMS Regional Conference on Random Number Generation. *Random number generation and quasi-Monte Carlo methods*, volume 63. SIAM, 1992. (cited page 143)

[136] The University of Tennessee, The University of California Berkeley, The University of Colorado Denver, and NAG Ltd. daxpy.f file reference, November 2011. (cited page 130)

[137] Vojin G Oklobdzija. An Algorithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(1):1–5, March 1994. (cited page 119)

[138] F Aleixo Oliveira. Interval Analysis and Two-Point Boundary Value Problems. *SIAM Journal on Numerical Analysis*, 11(2):382–391, April 1974. (cited page 45)

[139] William G Osborne, Ray CC Cheung, José Gabriel F Coutinho, Wayne Luk, and Oskar Mencer. Automatic Accuracy-Guaranteed Bit-Width Optimization for Fixed and Floating-Point Systems. In *Field Programmable Logic and Applications*,

2007. FPL 2007. International Conference on, pages 617–620. IEEE, 2007. (cited page 53)

[140] B Parhami. Computer Arithmetic, 2000. (cited pages 38, 39, 40, 41, and 165)

[141] Behrooz Parhami. *Computer arithmetic: algorithms and hardware designs*. Oxford University Press, Inc., 2009. (cited page 1)

[142] Douglass Stott Parker. *Monte Carlo Arithmetic: exploiting randomness in floating-point arithmetic*. Computer Science Department, University of California, 1997. (cited pages vii, xiii, 40, 55, 70, 72, 73, 74, 75, 77, 78, 83, 88, 89, 90, 92, and 94)

[143] Douglass Stott Parker. Monte Carlo Arithmetic, October 2003. http://www.cs.ucla.edu/~stott/mca/. (cited page 52)

[144] D.S Parker, B Pierce, and P.R Eggert. Monte Carlo arithmetic: How to Gamble with Floating Point and Win. *Computing in Science & Engineering*, 2(4):58–68, 2000. (cited pages 73, 74, and 78)

[145] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007. (cited pages 96 and 108)

[146] L B Rall. Representations of Intervals and Optimal Error Bounds. *Mathematics of computation*, 41(163):219–227, July 1983. (cited page 47)

[147] Arnab Ray and Raoul Jetley. Model-based development: a new approach to engineering medical software. *Biomedical Instrumentation & Technology*, 44(1):51–53, 2010. (cited page 3)

[148] Theodore J Rivlin. Chebyshev Polynomials. From Approximation Theory to Algebra and Number Theory. *Pure Appl. Math.(NY)*, 1990. (cited page 93)

[149] R Rojas. Konrad Zuse's Legacy: The Architecture of the Z1 and Z3. *Annals of the History of Computing, IEEE*, 19(2):5–16, 1997. (cited pages 5, 19, and 21)

[150] Saul Rosen. Electronic Computers: A Historical Survey. *Computing Surveys (CSUR*, 1(1), March 1969. (cited page 5)

[151] Howard H Rosenbrock. An Automatic Method for Finding the Greatest or Least Value of a Function. *The Computer Journal*, 3(3):175–184, 1960. (cited page 96)

[152] Bradley Moskowitz Russel E Caflisch. Modified Monte Carlo Methods Using Quasi-Random Sequences. 1995. (cited page 63)

[153] Mutsuo Saito and Makoto Matsumoto. Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer, 2008. (cited page 67)

[154] A Sandu. *CSE-690 Home Project 2*, November 2000. www.cs.vt.edu/∼asandu/Courses/MTU/CSE690/proj-2.ps. (cited page 95)

[155] M.J. Schulte and Jr. Swartzlander, E.E. A Family of Variable-precision Interval Arithmetic Processors. *Computers, IEEE Transactions on*, 49(5):387 –397, may 2000. (cited page 52)

[156] Rudolf Schürer. A comparison between (quasi-) monte carlo and cubature rule based methods for solving high-dimensional integration problems. *Mathematics and computers in simulation*, 62(3):509–517, 2003. (cited page 143)

[157] Claude E Shannon. A Symbolic Analysis of Relay and Switching Circuits. *American Institute of Electrical Engineers, Transactions of the*, 57(12):713–723, 1938. (cited page 6)

[158] R Skeel. Roundoff Error and The Patriot Missile. *SIAM News*, 1992. (cited page 38)

[159] Ilya M Sobol. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86–112, 1967. (cited pages 67 and 143)

[160] Pat H Sterbenz. *Floating-Point Computation*, volume 26. Prentice-Hall Englewood Cliffs, NJ, 1974. (cited page 90)

[161] J.E. Stine and M.J. Schulte. A Combined Interval and Floating Point Multiplier. In *VLSI, 1998. Proceedings of the 8th Great Lakes Symposium on*, pages 208 –215, feb 1998. (cited page 52)

[162] T Sunaga. Theory of an Interval Algebra and its Application to Numerical Analysis. *Japan Journal of Industrial and Applied Mathematics*, 2009. (cited page 46)

[163] Bonnie Toy and Will Menninger. C implementation of the linpack benchmark, February 1994. Available from http://www.netlib.org/benchmark/linpackc.new. (cited pages 96 and 123)

[164] Bruno Tuffin. On the use of low discrepancy sequences in monte carlo methods. *Monte Carlo Methods and Applications*, 2:295–320, 1996. (cited page 70)

[165] Jean Vignes. A stochastic arithmetic for reliable scientific computation. *Mathematics and computers in simulation*, 35(3):233–261, 1993. (cited page 51)

[166] Jean Vignes. Discrete stochastic arithmetic for validating results of numerical software. *Numerical Algorithms*, 37(1-4):377–390, 2004. (cited page 51)

[167] C. S Wallace. A Suggestion for a Fast Multiplier. *Electronic Computers, IEEE Transactions on*, (1):14–17, 1964. (cited page 10)

[168] GW Walster and D. Chiriaev. Interval Arithmetic Programming Reference: Forte TM Workshop 6 Update 1 C++. *Sun Microsystems Inc*, 2000. (cited page 52)

[169] Stefan Wegenkittl. Monkeys, Gambling, and Return Times: Assessing Pseudorandomness. In *WSC '99: Proceedings of the 31st conference on Winter simulation: Simulation - A Bridge to the Future*. ACM, December 1999. (cited page 63)

[170] James H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover Publications, Incorporated, 1994. (cited pages 2, 42, 89, and 94)

[171] JH Wilkinson. Error analysis of floating-point computation. *Numerische Mathematik*, 2(1):319–340, 1960. (cited page 2)

[172] Xilinx Inc. *LogiCORE IP AXI Interconnect v2.1*. http://xilinx.com/support/documentation/ip_documentation/axi_interconnectv2_1/pg059-axi-interconnect.pdf. (cited page 122)

[173] Xilinx Inc. *LogiCORE IP AXI Timer v2.0*. http://xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf. (cited page 122)

[174] Xilinx Inc. *LogiCORE IP Floating-Point Operator v7.0 Product Guide*. http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf. (cited page 114)

[175] Xilinx Inc. *LogiCORE IP Processing System 7*. http://www.xilinx.com/support/documentation/ip_documentation/processing_system7/v4_00_a/ds871_processing_system7.pdf. (cited page 122)

[176] Xilinx Inc. *LogiCORE IP Processor System Reset Module v5.0*. http://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf. (cited page 122)

[177] Xilinx Inc. *Vivado Design Suite - AXI Reference Guide*. http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf. (cited page 115)

[178] Xilinx Inc. *Zynq-7000 All Programmable SoC Overview*. http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. (cited page 113)

[179] Xilinx Inc. *Zynq-7000 All Programmable SoC Technical Reference Manual*. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. (cited page 113)

[180] R Yates. Fixed-point arithmetic: An Introduction. *Digital Signal Labs*, 2007. http://personal.atl.bellsouth.net/y/a/yatesc/fp.pdf. (cited page 15)

[181] JHC Yeung, EFY Young, and PHW Leong. A Monte-Carlo Floating-Point Unit for Self-validating Arithmetic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 199–208, 2011. (cited pages 52 and 73)

[182] Tjalling J Ypma. Historical development of the newton-raphson method. *SIAM review*, 37(4):531–551, 1995. (cited page 17)

[183] J Zilinskas and IDL Bogle. Balanced Random Interval Arithmetic. In *Computers & Chemical Engineering*, pages 839–851. UCL, Ctr Proc Syst Engn, Dept Chem Engn, London WCIE 7JE, England, 2004. (cited pages 46, 47, and 48)