CORE
Metadata, citation and similar papers at core.ac.uk
Provided by ScholarBank@NUS

ACCURACY-AWARE APPROXIMATIONS FOR ERROR ANALYSIS

ELAVARASI MANOGARAN
MSc in Computer Science, University of Edinburgh 2011

A THESIS SUBMITTED
FOR THE DEGREE OF MASTER OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2017

Supervisor:
Associate Professor Wong Weng Fai

Examiners:
Associate Professor He Bingsheng
Professor Tulika Mitra

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.
This thesis has also not been submitted for any degree in any university previously.

ELAVARASI MANOGARAN

1 April 2017

Acknowledgments

Firstly, I would like to thank the highly intellectual source who was guiding me throughout my life. I'm immensely grateful and thankful for providing me with the best in everything that I could think of. Next, I would like to extend my heartest thanks to the following members for making me who I'm today.

I would like to thank my supervisor Professor Weng Fai Wong for his steady support and encouragement offered to me in this work without which this journey wouldn't have been complete. I thank him for his unwavering patience and guidance and for making me realize that the choices I make can have a huge impact on my life. I would also like to extend my deepest gratitude to my co-supervisor and adviser Professor John L. Gustafson for offering me the opportunity to work on his breakthrough invention, unums. It has been an immense pleasure working with him and I thank him for the constant encouragement he offered me with every milestone.

I thank my parents for they have been there for me, whenever I needed them. I thank the Almighty for letting me choose the most loving parents and sister. The next most loving gift is my husband. Having him here with me is the best blessing I could ever ask for. Sailing through this journey together with him helped me to overcome the tough times, encouraging each other to pursue the higher perspectives of our life. I would also like to thank my grandparents and the entire family for supporting me through all of my academic pursuits. Finally I would like to thank my beloved friends Bayan and Iman for being with me all these years in times of joy and grief. Your company is highly appreciated and loved. These few years of transition would've been a tough call without you both. I would again like to thank the super power for helping me find you. I extend my warmest thanks to all my labmates for their collaboration, support and company offered to me during this period.

Thankyou all, this was a lovely journey!

Contents

1 Introduction **8**

1.1 Overview 8

1.2 Motivation and Aim 10

1.3 Content Overview 11

2 Background **13**

2.1 Floating Point Vs Fixed Point Designs 13

2.2 Multiple-Precision Floating Point and Interval Arithmetic: MPFR Vs MPFI 14

2.3 Universal Number Computations (UNUM Arithmetic) 15

2.4 Accuracy, Precision and Correctness 16

2.5 Rounding Errors 16

2.5.1 Relative Error and Signal-to-Quantisation Noise Ratio 16

2.5.2 ULP 17

3 Related Work **19**

3.1 Software Approximation Techniques 19

3.2 Bitwidth Analysis Techniques and Tools 20

4 Arbitrary Precision Bitwidth Analysis 23

4.1 Overview 23

4.2 Range and Precision Analysis 24

4.2.1 Analytical Range Analysis Methods: Interval Arithmetic Vs Affine Arithmetic 25

4.2.2 Profiling Based Range Estimation Utility 27

4.3 Arbitrary Precision Floating Point to Fixed Point Conversion 28

4.4 Case Studies 29

4.4.1 FIR and IIR Filters 30

4.4.2 16 Point FFT 30

4.4.3 8×8 DCT 31

4.4.4 Strassen's Matrix Multiply 32

4.4.5 Chebyshev Function Approximation 32

4.5 Results 33

5 Accuracy-Aware Computing 36

5.1 Rounding Errors and The Pitfalls of Conventional Computer Arithmetic Methods 36

5.2 Case Studies 37

5.2.1 Creeping Crud On A 20 Million Summation 37

5.2.1.1 Float's Roundoff Behind The Scenes 38

5.2.1.2 Compensated Summations 39

5.2.1.3 Extended Precision Floating Point and Interval Arithmetic 40

5.2.1.4 Unum Arithmetic 41

5.2.2 Non-Dyadic Fractions: Patriot Missile Crash 43

5.2.2.1 Comparison Of Floats, MPFR, MPFI and Unums . . 44

6 Conclusion and Future Work **46**

6.1 Future Work 47

Appendix A An Example of Float-to-Fixed Point Converted Code **55**

A.1 Strassen's Matrix Multiplication 55

Appendix B Unum Constraint Solver Code **61**

B.1 Solver for Robot Arm Inverse Kinematics 61

List of Tables

2.1 Comparison of the different number representations based on the total number of bits, fields used, maximum representable value, rounding modes used and overflow/underflow handling. HP - Half Precision, SP - Single precision, DP - Double precision, QP - Quadruple precision . . 18

4.1 Comparison of resources consumed and execution time (clock cycles x clock period) for single (SP), double (DP) and arbitrary precision (AP) implementations 34

5.1 Floating point Roundoff behaviour of the last 3222786 iterations of the 20 million summation code 38

5.2 Roundoff behaviour of the last 3222786 iterations of the 20 million compensated summation code 40

5.3 Roundoff behaviour of the last 3222786 iterations of the 20 million summation using Unum arithmetic 42

5.4 Arithmetic methods used, results and run time for the 20 million summation using 32-bits maximum vs 32-bits on average 43

5.5 Arithmetic methods used, results and run time for the Patriot missile simulation using 32-bits maximum vs 32-bits on average 46

List of Figures

2.1 Single precision IEEE 754 Floating Point format (top) and Fixed point format representation (bottom) 14

2.2 UNUM format with 6 different fields as depicted in the Unum computing book 16

4.1 C code for the FIR filter with original version (left) and the modified version produced by the range estimator (right) 27

4.2 Overview of the framework for float-to-fixed point conversion 29

5.1 C program for counting from 1 to 20 million over a loop 38

5.2 C program for 20 million summation modified to support compensated summation technique 39

5.3 C program for the internal clock counter of the patriot missile 44

Chapter 1

Introduction

1.1 Overview

Advances in computer hardware and software and the evolving parallel computing systems do not always provide a pleasing solution that satisfies the performance and energy constraints. Adding more application specific cores or designing powerful interconnects is not always the best solution for building powerful computer systems. Firstly the existing systems lack the necessary support for the evolving software, slowing down the application performance. Secondly these systems have already hit the power wall. Taking into account these constraints, a research that focuses on an alternative computing technology such as ***Approximate Computing*** would encourage an interesting shift in the way people use computers to accomplish different tasks [55].

The electrical power and storage demands has been the biggest obstacle for the most prevalent exascale computing applications. Approximate computing has become a promising solution to this problem through which the accuracy of specific error-resilient parts of these applications can be compromised in return for higher performance and power constraints. Approximate computing describes the transition from exact to imprecise computing to achieve performance improvements and energy efficiency. Not all computations need to be accurate. Approximate computing can benefit from this inexactness of data which is observed in wide range of applications such as computer

vision, machine learning, image processing, digital signal processing, search, sensors, recommendation systems etc. Mapping such error tolerant applications into unreliable hardware components such as approximate floating point units and memory can be achieved by using novel approximation aware programming models, algorithms and formal methods to support approximation.

Approximation came in different varieties: software level approximation techniques such as loop perforation, skipping tasks and memory accesses, memoization, neural network based accelerators, using faulty hardware; approximating memory components such as SRAM, DRAM and non-volatile memories; approximating various processor components, GPUs, FPGAs and so on. In general, approximate computing encompasses an integrated approach to developing techniques at various layers of the computing stack such as the circuit, architecture and software levels in addition to developing design methodologies and tools for synthesis and formal verification. Such a cross layer optimization technique has been showed to fully exploit the potential of approximate computing by achieving higher energy savings instead of restricting approximation at a single level [54] [6].

Bitwidth analysis is a software level approximation technique to realize the tradeoffs between application performance and resource constraints. We believe that such a software level bitwidth reduction technique coupled with high level synthesis and output sensitivity analysis can prove to be more efficient for predictive approximations when compared to the state-of-the-art bitwidth analysis techniques.

Alternative approaches to floating point approximation such as UNUM computing have evolved. UNUMs claims to improve the mathematical behaviour of computer systems by excepting from roundoff, overflow and underflow. In contrast to the existing approximation techniques, UNUMs are expected to improve the answer quality while simultaneously reducing the number of bits of precision needed for the computations. As an extension to our work on bitwidth analysis, we present case studies to analyse and compare some of the alternate computer arithmetic techniques and methods including multiple precision floating point, interval arithmetic and UNUMs in terms of their roundoff behaviour. We also discuss the work-in-progress UNUM constraint solver

which is capable of solving both floating point and integer constraints in the future work section. This suggests opportunities for error analysis and benchmarking in the area of global optimization and constraint solving.

1.2 Motivation and Aim

Applications are often assigned a precision which is much higher than what is required. Assigning a higher precision can produce more accurate results but drastically slows down the application whereas a lower precision might produce unacceptable results which can affect the application quality constraints. The challenge is to minimize the precision as much as possible besides producing *good enough* results, which satisfy the application error constraints. Several efforts in the past including [12, 15, 28, 29, 37, 48] have presented techniques for bitwidth analysis and optimization, coupled with sensitivity analysis of the output to achieve a reduced precision version of programs. Although bitwidth analysis came up as the most straightforward solution, the urge to come up with new energy efficient and performance centric designs has motivated researchers to study, evaluate and compare the efficacy of existing techniques for customizing the bitwidth [9] besides proposing new methods [26] [52]. Most of these techniques require identifying a suitable range and precision for the program variables, *formal verification* to verify the quality constraints of the application and *synthesis* to generate approximate circuits to evaluate the trade-offs between area, performance and accuracy of the designs.

We observe from the above-mentioned works, that bitwidth reduction has been well studied and experimented for either a Floating Point (FP) or a Fixed Point (FixP) setting based on the implementation of the chosen application. Floating point designs consumes much higher resources and energy compared to a fixed point counterpart [16]. Although emerging FPGAs have the ability to harness floating point operations, there are a number of applications that our approach is particularly relevant to, that do not require such a huge dynamic range used by the floating point operations so as to meet their quality constraints. For this reason we choose to convert floating point programs to fixed point equivalents to study their resilience to errors with respect to the observable

resource gains.

A unified approach of bitwidth analysis for float-to-fixed transformations have been proposed in [16]. Nevertheless, to obtain an optimal bitwidth, these techniques require mathematical algorithms evaluating the error tolerance of each node of a computation and its sensitivity with respect to the final output.This could be a complex process and is time consuming for programs dealing with huge number of inputs exploiting an arbitrary precision space. Instead of computing the error tolerance at each computational node, inspired by [31], we propose an end-to-end sensitivity metric which defines the acceptability requirements of the program outputs to be satisfied by both the FP and FixP designs. To summarize, the main objectives of this work are as follows:

- To identify a systematic approach for verifying the floating point to fixed point output error constraints through high level synthesis by leveraging the state of the art Xilinx Vivado HLS FPGA design tool.
- To evaluate the reduction in hardware resource consumption of arbitrary precision implementations through floating point to fixed point conversion.
- To compare and evaluate the floating point computations with multi-precision floating point, interval arithmetic and UNUM computations to make well informed choices of predictive approximations and automatic error analysis.

1.3 Content Overview

This work is organized in 6 chapters. The first chapter provides the reader with a general introduction to the evolution of approximate computing, the motivation to approximations and the major goals and approaches to the work. The second chapter covers the background of the different number formats that will be explored in the later chapters such as Floating point, Fixed point, Multiple precision Floating point and Interval Arithmetic and Universal Numbers. This is followed by some background to range and precision analysis and some basic definitions to accuracy, precision and correctness. Chapter 3 presents the related work on the existing software approximation techniques

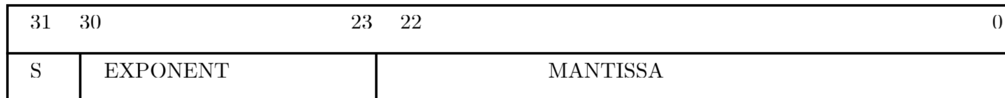
and tools for bitwidth optimization. Chapter 4 describes the proposed approach to arbitrary precision bitwidth analysis, Chapter 5 describes the case studies and compares and evaluates the results across different number systems. Finally chapter 6 provides a summary and conclusion for the study with approaches for future work.

Chapter 2

Background

2.1 Floating Point Vs Fixed Point Designs

The IEEE 754 FP representation is currently the widely used representation for real numbers. The two most common forms of this representation are the single and double precision formats. A single precision float occupies 1 sign bit, 8 bits of exponent and 23 bits of mantissa. It is to be noted that the exponent is a signed number and is represented using a bias method (for instance an excess-127 form is used for the single precision FP number). Whereas for a double precision float, the bitwidth requirements are much higher and goes up to 1,11 and 53 bits respectively. The representation of a single precision IEEE FP number is shown in Figure 2.1. Despite being widely used, floating point representation consumes a huge area when implemented in hardware since it has to process the exponent scaling and take care of rounding to ensure every FP operation performs correctly. Thus the FP units demand a major share of the power budget. An alternative to the FP design is the FixP representation which is more straightforward and overcomes the complexity of designing the FP units. Since all of the operations of a FixP computation involves integer values, and only little pre or post-normalization is required. Whereas, a FP arithmetic which involves tedious pre- and post- normalization, requires complex hardware such as priority encoders and variable shifters which incur large combinational delays and huge resource consumption [16].



$$FPvalue = (-1^{S}) \times 2^{EXPONENT} \times 1.MANTISSA$$

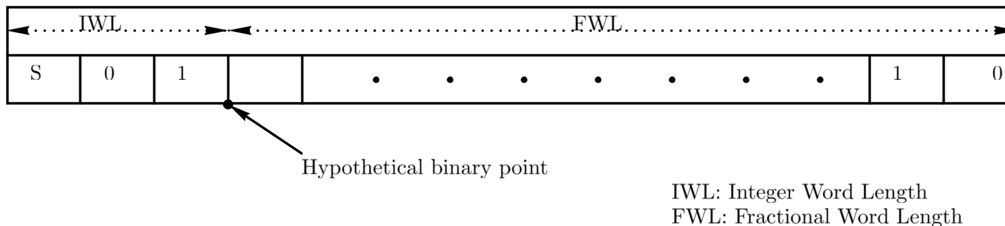


Figure 2.1: Single precision IEEE 754 Floating Point format (top) and Fixed point format representation (bottom)

2.2 Multiple-Precision Floating Point and Interval Arithmetic: MPFR Vs MPFI

GNU MPFR [14] is a multiple precision binary floating point library which extends IEEE 754 and is based on the GNU MP library. Compared to the other exhaustive list of multi-precision floating point libraries, we chose MPFR for this work because it is backed up by mathematical proofs of correctness and reliability [50] and provides the correct rounding for all the operations and mathematical functions. MPFR as an extension of the IEEE 754 standard produces results that are portable across different computer configurations and uses radix 2 of the form 2^k where k is the wordsize in bits. MPFR provides four rounding modes of the IEEE-754 standard (round to nearest, round towards zero, round towards plus infinity and round towards minus infinity) including away-from-zero and provides 53 bits of precision or mantissa giving the flexibility to reproduce all double precision computations. MPFR also allows the flexibility to modify the number of exponent and mantissa bits to any arbitrary number that the user deems good enough for his computations. The downside of this library is that it cannot keep track of the accuracy of the computation throughout the arithmetic operations. For instance if the operands are stored in a variable with limited precision and the result is stored in a variable with larger precision, then the library would still perform the computations with full precision which is reflected as a wastage of bits.

MPFI (Multiple-Precision Floating point Interval) [39] is a C library for implementing arbitrary precision interval arithmetic. With fixed precision floating point computations, interval arithmetic is known to suffer from the problem of wider and overestimated results even with the available machine precision. The goal of MPFI is to allow interval computations to be performed with higher precision. The endpoints of MPFI intervals are MPFR floating point numbers. MPFI provides specific in-built functions to bisect the intervals, finding midpoints, merging intervals etc, which makes it a suitable tool to implement and test applications like robotics and finding the real roots of a polynomial and other global optimization problems. This work uses both MPFR and MPFI libraries for evaluating both arbitrary precision floating point and interval arithmetic computations.

2.3 Universal Number Computations (UNUM Arithmetic)

Universal Numbers or UNUMs [20] is a superset of IEEE floating point and encompasses all IEEE floating-point formats as well as fixed-point and exact integer arithmetic. This number format contains meta fields to save storage and is expected to be less demanding on memory and bandwidth. UNUMs are designed to produce accurate results that do not round, overflow or underflow. Unums guarantee bitwise identical results across different computer architectures. Unlike the other number formats that define performance in terms of operations performed per second such as FLOPS, UNUMs introduce the idea of measuring performance in terms of the knowledge obtained about the answer. Figure 2.2 shows a preview of the unum format as depicted in [20]. The left three fields are the same as the IEEE floating point but differ in the way they represent NAN and infinity. The ubit or the uncertainity bit when set is used to represent an inexact number. The utag represents the fields ubit, exponent size and the fraction size that are unique to the unum format and helps to distinguish a unum from a float. Unlike the traditional interval arithmetic which can only represent closed intervals, UNUMs can represent both open and closed intervals. UNUM computations are found to solve problems across wide range of domains including polynomial equation solvers, physics simulations like n-body problems and galaxy colliders [20].

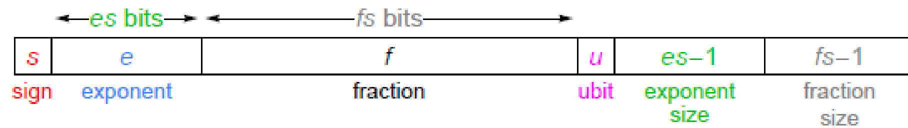


Figure 2.2: UNUM format with 6 different fields as depicted in the Unum computing book [1]

Table 2.1 at the end of this chapter summarizes the different number representations and their comparison based on the fields used, number of bits, maximum number range, rounding modes and overflow/underflow handling.

2.4 Accuracy, Precision and Correctness

Solving for the necessary precision analytically, to satisfy a desired accuracy is still a challenging area of research. While arbitrary precision library tools can help with this process, this is not the ultimate solution to the problem. We define the terms accuracy, precision and correctness of computations as follows,

- ***Accuracy*** is the closeness of the computed result to a correct result
- ***Precision*** is the total number of bits available to store a number
- ***Correctness*** is not necessarily being accurate or precise but confirming to an acceptable quality

2.5 Rounding Errors

2.5.1 Relative Error and Signal-to-Quantisation Noise Ratio

Relative error is the ratio of absolute error to the magnitude of the exact floating point value. Absolute error is the difference between the floating point number and

[1]The End of Error Unum Computing by John L. Gustafson

[2]Revol, Nathalie, and Fabrice Rouillier. ”Motivations for an arbitrary precision interval arithmetic and the MPFI library.” Reliable computing 11.4 (2005): 275-290.

the real number that it is approximating. In this work, we make use of the ***Signal-to-Quantisation Noise Ratio (SQNR)*** to evaluate the computation accuracy. We fix a minimum SQNR for our floating point precision analysis and also set a maximum allowable SQNR degradation that could be allowed when computing the precision of the fixed point computations. SQNR defines the ratio of the desired signal power to the quantisation noise power. The signal is the application output using double precision floating point arithmetic and the quantisation noise is the different between the actual double precision and the result generated by the fixed point code using truncation. This quantisation noise can in other words be called as the relative error. The SQNR for a sample signal y is represented as follows [1].

$$SQNR = 10log_{10} \frac{\sum_n y^2[n]}{\sum_(\hat{y}[n] - y[n])^2} \tag{2.1}$$

Since the applications we target for our experiments are all important routines from the Digital Signal Processing (DSP) domain, we believe that the SQNR metric is a good match for our evaluations.

2.5.2 ULP

The term ULP, referring to 'Units in the Last Place' is the most usual way of measuring the rounding error at least in conventional floating point arithmetic. The round-to-nearest mode of the floating point corresponds to an error of less than or equal to 0.5 ulp [18]. Consider a real number 4.35 approximated to 0.424×10^1 using base (β) 10 and 3 bits of precision(p). The error in terms of ulp is 0.5 and the relative error is 5.05ε where $\varepsilon == (\beta/2)\beta^{-p}$ as suggested by [18].

Table 2.1: Comparison of the different number representations based on the total number of bits, fields used, maximum representable value, rounding modes used and overflow/underflow handling. HP - Half Precision, SP - Single precision, DP - Double precision, QP - Quadruple precision

Format	Bit fields	Total bits	Maximum representable value	Rounding modes supported	Overflow/ Underflow handling
HP float	Sign, Exponent, Mantissa	16 (1 sign, 5 exponent, 1 implicit + 10 mantissa)	65504	Nearest even (default), Towards $+\infty$, Towards $-\infty$, Towards zero	Program aborts with a $+\infty$ or $-\infty$
SP float	Sign, Exponent, Mantissa	32 (1 sign, 8 exponent, 1 implicit + 23 mantissa)	3.4028×10^{38}	Nearest even (default), Towards $+\infty$, Towards $-\infty$, Towards zero	Program aborts with a $+\infty$ or $-\infty$
DP float	Sign, Exponent, Mantissa	64 (1 sign, 11 exponent, 1 implicit + 52 mantissa)	1.7976×10^{308}	Nearest even (default), Towards $+\infty$, Towards $-\infty$, Towards zero	Program aborts with a $+\infty$ or $-\infty$
QP float	Sign, Exponent, Mantissa	128 (1 sign, 15 exponent, 1 implicit + 112 mantissa)	1.1897×10^{4932}	Nearest even (default), Towards $+\infty$, Towards $-\infty$, Towards zero	Program aborts with a $+\infty$ or $-\infty$
MPFR	Same as float (with variable exponent and mantissa size)	Variable upto 64 (53 bit significand)	Same as floats based on precision used	All rounding modes supported by floats + away from zero	Program aborts with a $+\infty$ or $-\infty$
MPFI	Same as float (Supports MPFR float types as left and right interval endpoints)	Variable based on computer memory (Largest precision bits reported: 4095 [2])	Same as floats based on precision used	All rounding modes supported by floats (Most prevalent: Towards $+\infty$, Towards $-\infty$)	Resultant interval is bounded by $+\infty$ or $-\infty$
Type I Unums	Sign,Exponent, Fraction,Ubit, Exponent size, Fraction size	Variable based on computer memory	6.805543 10^{38} for a 32-bit unum (approx. twice as large as 32-bit floats)	No Rounding	Resultant ubound is bounded by maximum or minimum possible real number supported by a chosen exponent and fraction size.

Chapter 3

Related Work

This section will provide a review of the previous work on approximate computing, the types of approximation, floating point precision analysis and bitwidth optimization of fixed point programs. The main target of approximate computing is to exploit the trade-off between accuracy and performance or energy consumption.

3.1 Software Approximation Techniques

Approximation techniques come in different flavours. Nevertheless, the main goal of approximation has been described by the efforts to identify error resilience in applications that could subsequently result in energy and resource savings and/or performance improvements. Several related works have proposed both dynamic and static variants of program analysis techniques to analyse the criticality of applications. In [40], Rinard presents a dynamic approach that allows to identify and skip execution of selected resilient or non-critical tasks during the program execution. If the distortion produced by the task-skipping is unacceptable, the task is marked as critical and non-critical otherwise. Following this in [41], the same author presents an early stage termination for eliminating idle processors during a parallel computation. This technique takes advantage of the performance improvements obtained by skipping a parallel task phase whilst also producing acceptable results. The above techniques make use of probabilistic reasoning to characterize the accuracy constraints of task based computations.

In [3, 45] several static techniques such as Green and EnerJ were proposed to enable approximation at a finer granularity than tasks. This is achieved by approximating expensive functions and loops using early loop termination. Similarly EnerJ proposes approximate type specifiers to automatically map approximate variables to low power storage. A common idea in all of these techniques and others [22, 46] is skipping tasks or loops to improve performance. However, the error induced by these techniques is not easy to predict. [6] proposes Rely, a language for expressing and analysing approximate computations to be run on unreliable hardware components. We observe that the first and the foremost step of implementing the above mentioned software-level approximation techniques is to identify the critical regions of the program that have a huge significance on the output of the application and distinguish them from the non-critical regions of the application that either do not affect or have a very less impact on the overall output. Recent efforts like [31], reduce the approximation effort by automatically selecting the approximate instructions and data that could be stored in approximate memory components. Although the approximation design effort is much reduced these days, it is observed that the level of granularity in which these approximation occurs is mostly restricted to the instruction level programming constructs. One alternative direction is in the reduction of floating point precision using techniques such as code profiling or search algorithms [5, 10, 43, 44] and [27].

3.2 Bitwidth Analysis Techniques and Tools

This section describes some of the existing state-of-the art bit-width analysis techniques and tools used for both Floating point and fixed point applications.Optimizing the bitwidth of the Floating Point(FP) mantissa has been observed as the most efficient way of reducing the power consumption of the FP data elements [51] but this requires replacing each FP operation with a software emulation of a reduced bitwidth FP unit or the use of parametrized floating point libraries [47] to measure the program accuracy and area requirements. Results from [8] shows that if redundant bits are identified and reduced, the hardware resource cost can be simplified. The same paper also reports that the adders area is proportional to its bitwidth and multipliers area is proportional

to the product of its two input bitwidths using state-of-the art Altera stratix FPGAs. Except for a few system level languages like SystemC [19] and SpecC [17], other high level languages lack the facility to specify arbitrary precision bitwidths for variables and operations.

Fixed Point (FixP) computations are easier to synthesize but this format is not used widely and is often claimed to be difficult to design compared to the existing floating point applications. Addressing this issue, techniques like [16] have proposed a unified approach of bitwidth analysis for float-to-fixed transformations. This approach is useful compared to the previous approaches since it provides a systematic way of bitwidth analysis for both floating point and fixed point implementations, which aims to find if either the FP or the FixP is the best setting for a specific application. To determine an optimal bitwidth, these techniques require mathematical algorithms like automatic differentiation evaluating the error tolerance of each node of a computation and its sensitivity with respect to the final output. These analytical error models were used to express the output precision as the function of quantisation error of input and intermediate signals. This could be a cumbersome process and is time consuming for programs dealing with huge number of inputs exploiting an arbitrary precision space. Observations from the previous works [16], show that until a broader range of data inputs (say from 10^1 to 10^{12}), users can confidently choose to use a FixP implementation for a wide variety of applications, above which a floating point implementation would become more area efficient [16]. This observation greatly supports the need for a float-to-fixed conversion technique such as the one we have proposed, to facilitate accuracy-aware bitwidth analysis and to study the effects of different levels of output quality degradation that could lead to optimal levels of performance and resource gains.

In addition to that, techniques and tools such as [29, 48] use traditional interval arithmetic and affine arithmetic methods to analyse the range of floating point variables. In [57], the authors propose an Extreme Value Theory(EVT) based range estimation method for finding the maximum and minimum extremes of the observed range of the program inputs. Most of these techniques use a more conservative bound on the range estimation. In [44], a tool for floating point precision search is proposed. Although this tool suggests the type configurations that could be lowered in precision, they only work

on standard precisions such as float, half or double. This work takes into consideration the arbitrary precision configurations in addition to the standard ones. Our synthesis makes use of arbitrary fixed point data types provided by Xilinx Vivado HLS design tool. We use this tool to perform C++ simulations of the fixed point outputs and to emulate the variable bitwidth implementation precisely in the hardware.

Although there exists a growing body of tools and techniques for bitwidth reduction, most of these techniques work on automatically detecting the best mix of single or double precision for the code. Our technique differs from these techniques in being able to tune for an arbitrary number of precision bits for both the floating point and fixed point computations. This effectively means that the program variables are assigned the least possible precision required for satisfying user specified accuracy bounds.

Chapter 4

Arbitrary Precision Bitwidth Analysis

4.1 Overview

Existing works have studied bitwidth reduction from either a floating point or fixed point standpoint. Mostly these works address a standard precision or determine the best mix of standard single or double precision. As we propose to perform an integrated floating point to fixed point precision tuning, we identify a systematic approach for verifying the floating point to fixed point output error constraints. Using this approach in conjunction with high level synthesis tools we aim to evaluate the reduction in hardware resource consumption of arbitrary precision implementations.

Inspired by the potential benefits of fixed point implementations in terms of hardware cost, speed etc. which were discussed in the previous sections, we chose to convert specific floating point DSP routines into a reduced bitwidth fixed point implementation. A fixed point conversion, if not sensibly designed can suffer from overflows and lead to unacceptable results if sufficient wordlengths are not assigned to the inputs. Given a representative dataset satisfying a specific dynamic range of inputs and a user-specified SQNR error metric, our approach helps choose a suitable arbitrary precision configuration for fixed point bitwidth that has the potential to achieve reduced latency and

hardware resources. To achieve this, we studied the importance of both range and precision analysis methods for bitwidth optimization. Identifying the range of floating point input is essential to obtain the Integer WordLength (IWL) of the corresponding fixed point representation. The problem of range analysis involves studying the data dynamic range of all the inputs involved in the computation and ensuring that all of these inputs will be allocated enough bits to accommodate this range. Range analysis should also be able to ensure that no overflow occurs when mapping the computed IWL onto a fixed point design. Even in the case of fixed point designs, this should ensure that there are no redundant bits allocated [56]. Precision analysis involves analyzing the sensitivity of the output with respect to the slight changes introduced to the input [28]. To be more specific, this involves identifying the minimum mantissa bitwidth of all floating point variables in the computation that satisfies a given output sensitivity.

4.2 Range and Precision Analysis

Range and precision analysis are methods to determine the integer and fractional wordlength of both the FP and FixP numbers. The bitwidth (BW) of a FP or a FixP number is the sum of its Integer WordLength (IWL) and Fractional WordLength (FWL) [16] denoted as follows,

$$BW = IWL + FWL \tag{4.1}$$

This work focuses mainly on range analysis, more specifically we propose an automatic range estimator written in C for determining the IWL for the float -fixed point conversion. The range estimator is based on an existing simulation based range estimator [25] implemented using the C++ language. IWL is calculated using the dynamic range information computed by measuring the mean, variance and standard deviation of the variables. The dynamic range is estimated using the relation,

$$Range(x) = max\{(|\mu(x)| + n \times \sigma(x)), Amax|x|\} \tag{4.2}$$

where $\mu(x)$, $\sigma(x)$ and $Amax|x|$ is the average, standard deviation and absolute maximum value, respectively, of a given variable x. Unlike [25] which uses a larger value of n that overestimates the range, we set $n = 1$. Our measured ranges for up to 1000 random numbers generated within our predefined input range showed that no overflow will occur.

The IBW of the fixed point designs are calculated from the range results using,

$$IBW_x = \lceil log_2(Range(x)) \rceil + \alpha \tag{4.3}$$

$$\text{where } \alpha = \begin{cases} 1, mod(log_2(x_{max}), 1) \neq 0 \\ 2, mod(log_2(x_{max}), 1) = 0 \end{cases}$$

where x_{max} is the maximum value of the variable x observed during profiling [28]. The precision results obtained from the search are used as the FBW.

4.2.1 Analytical Range Analysis Methods: Interval Arithmetic Vs Affine Arithmetic

Interval Arithmetic (IA) [33] was invented by Moore in 1960s to solve range analysis problems. In this method each input signal is represented using a range comprising of maximum and minimum values of that number, indicating that the true value lies somewhere in between this range. This method works by obtaining the output dynamic range from the input signal dynamic range using a worse case propagation rule which assumes that the range obtained by this method varies independently over the given intervals. Hence mostly the IA method determines a range value which is a huge overestimation of the observed true value of the output range. Affine arithmetic methods works by calculating the dependency between the intermediate variables and the rounding errors propagated during computations such as loop iterations. This is more practical for systems which respect the rounding error dependency more than the actual input dependency on the output of the computation. For instance, a signal x is represented using the interval $\bar{x} = [x_{min}, x_{max}]$ where x lies between the minimum range value x_min and the maximum value x_max. The range of any arithmetic operation using interval

arithmetic is thus represented as a propagation function calculated using the maximum and minimum values of the operand intervals. The equations shown below represent the addition and subtraction operations performed using the interval arithmetic.

$$\overline{x}+\overline{y}=[x_{min}+y_{min},x_{max}+y_{max}] \tag{4.4}$$

$$\overline{x}-\overline{y}=[x_{min}-y_{max},x_{max}-y_{min}] \tag{4.5}$$

It can be seen that when an expression of the form, x is evaluated, we get the interval range twice as wide as the original interval x which instead lies in the range [0,0]. For a very long computation chain, these range estimates cause intolerable error accumulation. For this purpose, the Affine Arithmetic (AA) [11] based range analysis methods were introduced. Affine arithmetic differs from the IA by keeping track of correlations between the input signals on the round-off errors. For instance the signal x is represented in the affine form $\hat{x}$ as follows,

$$\hat{x}=x_0+x_1\varepsilon_1+x_2\varepsilon_2+....+x_n\varepsilon_n \tag{4.6}$$

where, $\varepsilon_i=[-1,1]$ ε_i is the independent uncertainty factor that contributes to the total uncertainty of the signal x. This involves converting an interval range of IA say $\hat{x}$ in the form $\overline{x}=[x_{min},x_{max}]$ into an affine form $\hat{x}=x_0+x_1\varepsilon_1$ where,

$$x_0=\frac{x_{max}+x_{min}}{2} \quad \text{and} \quad x_1=\frac{x_{max}-x_{min}}{2} \tag{4.7}$$

Thus if x takes the affine form $\hat{x}$ the previous overestimation by IA now becomes $\hat{x}-\hat{x}=0$ using the AA method. Although AA methods represent uncertainty of the signals in terms of linear approximations, the width of the range of performing these operations follows tighter bounds compared to the IA methods yet follows a conservative approximation for operations that are not in affine forms. For this reason, sometimes AA based range estimation could perform worse than the IA for typical scenarios as described in [28]. Other methods such as L1 norm, polynomial approximations used in Taylor series methods and polynomial dependence as described by [35] were presented and analyzed in an effort to further decrease the width of the range.

```
1  float fir(float input){
2    int i;
3    float h[32]={0.2234,
4    0.844,    ..-0.97656};
5    x[0] = input;
6    acc = x[0] * h[0];
7    for(i = 31; i>0; i--) {
8      acc = acc + x[i]*h[i];
9      x[i] = x[i-1];
10   }
11   y = acc;
12   return y;  }
```

```
1  float fir(float input){
2    int i;
3    float h[32]= {0.2234, -0.844,
4       ..-0.97656};
5    assign_range(&x_track, input);
6    count[index_lookup("x_track")]++;
7    assign_range(&acc, x[0] * h[0]);
8    count[index_lookup("acc")]++;
9    for(i = 31; i>0; i--)  {
10    add_range(&acc,acc,(x[i] * h[i]));
11    assign_range(&x_track, x[i - 1]);
12    count[index_lookup("x_track")]++;
13   }
14   assignment_range(&y, acc);
15   count_vars[index_lookup("y")]++;
16   return y;}
```

Figure 4.1: C code for the FIR filter with original version (left) and the modified version produced by the range estimator (right)

4.2.2 Profiling Based Range Estimation Utility

Besides much appreciation received by the above mentioned analytical methods, we chose to use a simulation based range estimation method for two main reasons. Firstly the analytical methods have been proved to be difficult to implement for linear time-invariant systems which has direct applications in signal processing, seismology, control theory and image processing where opportunities for approximation are abundant.

It is to be noted that analytical methods for range estimation are difficult to be applied for such applications which involves cyclic graph traversal over recursive structures accounting for their range propagation rules [25]. Secondly, these methods, guaranteeing the absence of overflows generally prove to be complex, and produce conservative bounds on output which might lead to over-estimation of the hardware resources specifically in settings which we are concerned in this work. Hence we built an automatic software range estimator based on a statistical scaling procedure similar to [25]. In this previous work, the authors use a C++ operator overloading feature to initiate a new data class for tracing the maximum value of an input signal. Unlike this method which requires manual modification of the float declaration, with range estimating C++ class, our range estimator works automatically for a range of programs tested in this work. Moreover current software implementations try to avoid operator overloading since it makes the code difficult to maintain. In [24], a C based range estimator which uses SUIF compiler support has been presented.

In our work, we developed a complete C implementation of the range estimator making use of source to sink style function calls for tracking the dynamic range of variables after every assignment or arithmetic operation. We use the python pycparser to automatically convert the original floating point program into a version used for the range estimation. We use this version to obtain the IWL of the fixed point variables. The range estimation program contains several functions that helps determine the dynamic range of the variables. Firstly the maximum and minimum ranges of the variables are obtained. This information is used to obtain the absolute maximum value which is required for the IWL determination. After every assignment or arithmetic operation, a corresponding source-to-sink, two operand style function as shown in the modified version (right) of Figure 4.1 is called, which performs the necessary operation associated with the function call. Every function, in addition to tracking the current value of the variable, updates the maximum and minimum value of output observed for that operation and computes the absolute maximum value of the output at that instance. In addition to that each function keeps track of the mean and standard deviation of the variable observed during the operation which is used to calculate the range of every variable. However, we realize that in order to use the proposed method, the user should specify a representative set of realistic inputs for range estimation as is the case with other simulation based methods. But in most of the cases, it is non-trivial for the user to guess the inputs. So we also employ a statistically guided refinement to find the worst case input for any given program [21] where we use average Signal-to-Quantisation-Noise ratio (SQNR) as a metric to analyze the quality of the outputs. Among the inputs tested if 95% of them produce at least as good as the required SQNR value, we consider them valid. The inputs are randomly generated using different seed values of the random number generator.

4.3 Arbitrary Precision Floating Point to Fixed Point Conversion

Our framework as shown in Figure 4.2 takes as input the original floating point program. The problem of float-to-fixed point conversion is divided into two major meth-

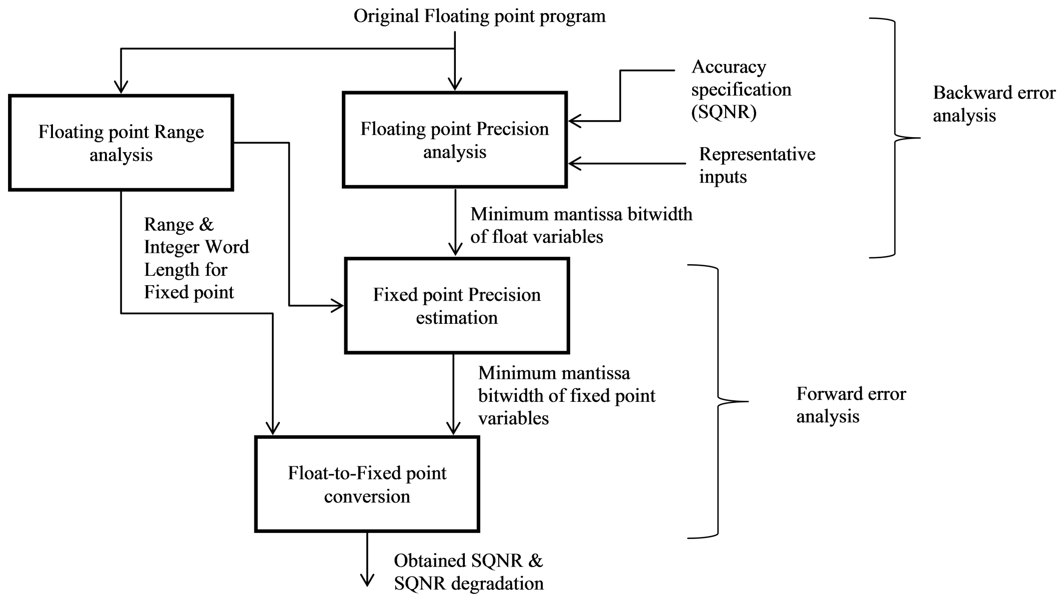


Figure 4.2: Overview of the framework for float-to-fixed point conversion

ods namely precision and range analysis. Precision analysis includes the problem of identifying the minimum mantissa bitwidth configuration of the input floating point variables. We make use of backward error analysis technique for precision estimation through which the user specifies the accuracy specification (SQNR error metric) to be satisfied at the program output, which can be used to identify the minimum bitwidth information for the inputs. The precision analysis is a parallel research conducted by my group. Range analysis is carried out using the statistical scaling technique as explained earlier. We calculate the data dynamic range of the variables along with the input ranges and the absolute maximum values of the intermediate values and output. Using this dynamic range information, the IWL to be used by the fixed point implementation can be obtained. In addition to the IWL, we also obtain the average exponent, i.e, the exponent of the average dynamic range of each variable that is observed during the simulation. A sample code for floating point to fixed point conversion for Strassen's matrix multiplication is listed in Appendix A.

4.4 Case Studies

Our synthesis and evaluations make use Xilinx Vivado HLS tool using Kintex-7 Xc7k160tfbg484-1as the target device. In this section we describe the various programs used to imple-

ment our bitwidth analysis approach.

4.4.1 FIR and IIR Filters

The FIR (Finite Impulse Response) filter as shown in [4] is one of the important digital filters used in DSP applications whose impulse response is finite. For a filter of order N, the operation of FIR filter can be expressed by the following relation,

$$y[n] = \sum_{i=0}^{N} h_{ix}[n-i] \tag{4.8}$$

where $x[n]$ is the input signal, $y[n]$ is the output signal, h_i is the value of impulse response of the filter at the ith instance.

An FIR filter of the above form has no feedback. This ensures that the rounding errors are not accumulated every time the loop is executed and the same amount of relative error occurs across calculations of different loop iterations.

The main difference between the FIR and IIR filters are, the impulse response of the FIR lasts for a finite duration of time as opposed to the Infinite Response Filters (IIR). The IIR kernel is a N-cascaded Biquad IIR with M points. Systems with impulse response functions which are non-zero over an infinite amount of time are called IIR systems. The IIR filters uses less amount of coefficients or taps than the FIR using recursion. This states that IIR filters use less amount of hardware or software resources as compared to the FIR filters but IIR proves to be unstable when processing high frequency components and hence it requires more fine-tuning to achieve the same result of the FIR filters.

4.4.2 16 Point FFT

The Fast Fourier Transform (FFT) is an application which provides an efficient way for calculating the Discrete Fourier Transform (DFT). The application is an N radix-2 FFT butterfly. The working of an FFT program mainly focuses on the operation of complex numbers which usually consist of a complex part and an imaginary part which

are specifically represented as signal, point, sample, value etc. in the jargon of complex notation. The operation of FFT is explained in the following steps,

- Decomposition of an N point time domain signal into N time domain signals each consisting of a single point.
- Determination of the N frequency spectra corresponding to the N time domain signals.
- Synthesis of the N frequency spectra into a single frequency spectrum.

We use the C version of the function to compute the 16 point real Fast Fourier Transform using the split radix algorithm provided by MIT [30]. The split radix algorithm uses both a radix-2 and radix-4 decomposition in the same FFT algorithm, exploiting the idea that the even numbered points of the DFT can be computed independently of the odd numbered points. The code requires 79 adds and 10 multiplies. A more detailed description of the algorithm can be found in [30].

4.4.3 8 × 8 DCT

DCT is a Fourier related transform that transforms a signal or image from the spatial domain to the frequency domain and uses only real numbers. DCT and FFT differ from the fact that DCT can approximate lines with fewer coefficients. The output array of DCT functions contains integers in the range -1024 to 1023. The DCT input is a 8 × 8 integer array which represents each pixel's gray scale level. The general equation for a 2D (N × N data items) DCT [42] is expressed as follows,

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i)C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) cos\left(\frac{(2x+1)i\pi}{2N}\right) cos\left(\frac{(2y+1)j\pi}{2N}\right) \tag{4.9}$$

$$C(x) = \begin{cases} \frac{1}{2}, & \text{if } x = 0 \\ 1, & \text{if } x > 0 \end{cases}$$

The DCT implementation used in this work uses 5 multiplies and 29 adds.

4.4.4 Strassen's Matrix Multiply

The 2 $\times$ 2 matrix multiplication as described by Strassen [49] is a commonly used processing element in most of the large matrix multiplication applications. We assume that the input matrix elements are within the dynamic range of [0,1] for our evaluation.

The operation of a 2 $\times$ 2 matrix multiplication is shown below,

$$\begin{pmatrix} q_{00} & q_{01} \\ q_{10} & q_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} \tag{4.10}$$

The four quadrants of the resultant matrix are calculated as follows,

$$\begin{aligned}
q_{00} &= p_0 + p_3 - p_4 + p_6 & p_0 &= (a_{00} + a_{11})(b_{00} + b_{11}) \\
q_{01} &= p_2 + p_4 & p_1 &= (a_{10} + a_{11})b_{00} \\
q_{10} &= p_1 + p_3 & p_2 &= a_{00}(b_{01} - b_{11}) \\
q_{11} &= p_0 + p_2 - p_1 + p_5 & p_3 &= a_{11}(b_{10} - b_{00}) \\
& & p_4 &= (a_{00} + a_{01})b_{11} \\
& & p_5 &= (a_{10} - a_{00})(b_{00} + b_{01}) \\
& & p_6 &= (a_{01} - a_{11})(b_{10} + b_{11})
\end{aligned}$$

4.4.5 Chebyshev Function Approximation

We used the the C++ release for Chebyshev approximation of the function sin(x) by J-P Moreau [34]. The Chebyshev coefficients are in the range (-1,1) and the integral of the function sin(x) is approximated by using the range of x in the interval [0, π]. The Chebyshev approximation involves three main functions. Given the function sin(x), lower and upper limits of the interval [0, π] for x, and a maximum degree N (we chose a value of 10), the first routine computes the N Chebyshev coefficients, such that sin(x) is approximated by N . Using the Chebyshev coefficients obtained from the former

routine, the second function returns the array of Chebyshev coefficients of the integral of the function. Finally the Chebyshev evaluation of the function sin(x) is evaluated using the above computed coefficients of the integral.

4.5 Results

We evaluated the hardware resource consumption using six case studies as described earlier by converting the floating point to fixed point designs targeting the Xilinx Kintex 7 xc7k160tfbg484-1 FPGA. The floating point to fixed point conversion is done in Vivado HLS [13] C++ arbitrary precision library which provides built-in arbitrary precision datatypes with various rounding and saturation modes for customizing the bitwidths of fixed point programs. The C versions of the programs are first converted to C++ so as to use this library. For FFT16, we use the input dynamic range of (-1, 1) and the input to the 2×2 matrix are in the range (0, 1). For function approximation, we implemented the approximation of the function sin(x) for using a degree 10 Chebyshev approximation. The Chebyshev coefficients and the value of x are in the range (-1, 1) and $[0, \pi]$ respectively.

We implemented an automatic float-to-fixed point conversion using macro generation and substitution. The SQNR error metric is used to evaluate the accuracy of the arbitrary precision output against the original double precision code since it is the most commonly used metric for floating point to fixed point conversions [25] [1]. Unlike most of previous works [29] [7] that aims to achieve a higher SQNR value by using higher target precisions, our aim is to reduce the resource consumption whilst staying within a given accuracy bound for the results. Most previous works used an average SQNR value of 60 to 80dB. For this reason, we fixed the error bound of our floating point precision search to be within these two values. After conversion to fixed point, we obtained an SQNR of around 56 to 65 dB. The FPGA resource consumption and the execution time for the double precision floating point versus the arbitrary precision fixed point versions are presented in Table 4.1.

The main observations from our FPGA synthesis results are,

Table 4.1: Comparison of resources consumed and execution time (clock cycles x clock period) for single (SP), double (DP) and arbitrary precision (AP) implementations

Case study	Hardware resource consumption				Execution time (ns)			SNR (dB)
	HW	SP	DP	AP	SP	DP	AP	
FFT16	DSPs	72	90	10	430	437	182	62
	FFs	7991	8351	1356				
	LUTs	8150	10014	3365				
Chebyshev	DSPs	19	17	2	1269	1742	1019	65
	FFs	4186	4084	647				
	LUTs	5559	5661	1096				
2x2 Matrix multiply	DSPs	24	62	7	187	233	61	60
	FFs	2761	5495	366				
	LUTs	2389	6564	813				
FIR filter	DSPs	5	14	2	2346	3149	1039	61
	FFs	543	1097	98				
	LUTs	476	1213	171				
IIR filter	DSPs	2	3	1	31	35	8	63
	FFs	231	450	20				
	LUTs	215	782	98				
8x8 DCT	DSPs	31	79	10	5198	6178	2958	56
	FFs	3395	6744	1833				
	LUTs	3474	8843	4604				
Average	all	6587	9927	2554	1577	1962	877	62
Ave. AP Impr. (%)		66%	82%		53%	62%		

- The arbitrary precision implementation on an average uses 78% and 68% lesser LUTs compared to the double and single precision versions, respectively.

- The number of DSPs consumed by the double precision implementations is large. This behaviour is expected since a single 32-bit multiply by itself requires more than one DSP [13]. Compared to the double precision version, our synthesis results for arbitrary precision showed up to 89% reduction in the number of DSPs for a similar output error specification. Interestingly, for two out of three applications, the percentage reduction in the number of DSPs for the single precision versions is quite similar to that for the double precision, indicating that the single precision is just as resource hungry.

- The execution time on an average is reduced to about 58% and 48% of the double and single precision version, respectively.

We further compared our area savings against a target precision of 16 bits (half precision). To do this, we computed the maximum fraction bitlength that could fit in a 16 bit wordlength after deducting the integer wordlength obtained from the range results. Our results show an average of 9% reduction in resources over the half precision results. Moreover, in comparison with half precision, for a similar output error specification, our results show upto 4% reduction in area for one of the programs 2×2 matrix, which is comparable to the existing bitwidth optimization methods [7,28]. This is the evidence that even the use of (uniform) half precision can be wasteful for certain applications.

Chapter 5

Accuracy-Aware Computing

5.1 Rounding Errors and The Pitfalls of Conventional Computer Arithmetic Methods

Error analysis, especially roundoff error analysis of computations is a favourite area yet not so touched voluntarily by the programming community. The only reason these errors are ignored is because it is hard to detect the source of these errors, i.e, where in a program these errors originate and what causes these errors. Even worse, most of these rounding errors werent recognized to be the real cause of erroneous computations, rather is jeopardised for lack of precision or accounted to be the result of any one of the several accepted anomalies of the IEEE floating point standards [32]. While rounding errors anomalies are ignored for the fact that they occasionally affect or almost dont have a bigger consequence on conventional computations (like gaming or graphics), they do really matter and might prove to be catastrophic for a wide category of computations, yet fractious to be analysed.

The focus of this section is to study the effect of roundoff errors in conventional computer arithmetic techniques involving long-running computations. In computations involving long running loops, accumulation of roundoff errors can result in not even near close results. Such behaviour is very much catastrophic to be ignored. One familiar example was the failure of the patriot missile in 1991 [2] which killed about 28 american

soldiers and injured around 100 people, all that because of gradual accumulation of rounding errors in the missiles internal clock which counted time in tenths of seconds. In the late 90s, there were embarrassing incidents reporting wrong exam results because of rounding errors [36].

Although computer systems have evolved at a faster rate till now, computer users, programmers and hardware designers are accustomed to the same traditional ways of arithmetic for several decades and have unfortunately not yet evolved adequately to assess and debug the root cause of errors and several numerical anomalies. While various techniques and tools exist to compensate errors and also to establish the tradeoff between accuracy of the results and obtained performance, there is no clear way of benchmarking the way these approximations are carried out. In this section, we use selected case studies to compare and analyse the rounding errors introduced by some of the conventional computer arithmetic techniques, which points out the importance of error analysis and the need for efficient benchmarking of approximations.

5.2 Case Studies

We compare and analyze the round-off behaviours of floats, compensated summation, multi-precision floating point and interval arithmetic and unum arithmetic through the following case studies. All experiments were carried out using Intel Xeon CPU E5-2680@2.70GHz.

5.2.1 Creeping Crud On A 20 Million Summation

Let us look at a simple loop as shown in Figure 5.1 that adds up numbers from 1 to 20 million (2×10^7). In the first look, it would seem like this number is very small compared to the maximum range of numbers the IEEE floating points can represent (the range $(2-2^{23}) \times 2^{127} \approx 3.402823 \times 10^{38}$) and in fact the final expected sum of 20 million is exactly representable using a IEEE 32-bit float.

```
float sumTest () {
float sum;
int i;
sum = 0.0;
for (i = 1; i <= 20000000; i++)
  {sum = sum + 1.0;}
printf ("%f\n", sum);
}
```

Figure 5.1: C program for counting from 1 to 20 million over a loop

Table 5.1: Floating point Roundoff behaviour of the last 3222786 iterations of the 20 million summation code

Iteration	Floating point mantissa $\times 2^{exponent}$ (The below number is entered to help count digits. The symbol ↓ is used to mark the maximum limit of mantissa above which the number is usually rounded) 1.12345678901234567890123↓1234	Result	Remark
16777215	1.11111111111111111111111↓ $\times 2^{23}$	16777215	Correct
16777216	1.11111111111111111111111↓0$\times 2^{24}$	16777216	Correct
16777217	1.00000000000000000000000↓1$\times 2^{24}$	16777216	Round nearest even Wrong!
16777217	1.00000000000000000000001↓1$\times 2^{24}$	16777218	Round-up, Wrong!
⋮	(behaviour repeats as that of iteration 16777217)	16777216	Wrong!
19999999	1.00000000000000000000000↓1$\times 2^{24}$	16777216	Wrong!
20000000	1.00000000000000000000000↓1$\times 2^{24}$	16777216	Wrong!

5.2.1.1 Float's Roundoff Behind The Scenes

The C program above, produces a result of 16777216. What caused the error was not because IEEE floats cannot represent numbers this large. From what we know, a single precision produces an overflow only when the numbers exceed $(2-2^{23}) \times 2^{127}$ maximum limit. The problem is because floating point rounds to nearest even, in other words, to the nearest multiple of 2 when the numbers are in the range of [-33554432,-16777217] or in [16777217,33554432]. What happens precisely during the last 3222786 iterations of the above code is illustrated in Table 5.1. One can see that when the single precision float adds upto the number 16777216 (until which it is exactly

```
float sumTest () {
float sum,comp,oldsum;
int i;
sum = 0.0; comp=0.0; oldsum=0.0;
for (i = 1; i <= 20000000; i++)
{comp=comp+1.0;
oldsum=sum;
sum=oldsum+comp;
comp=(oldsum-sum)+comp;}
printf ("%f\n", sum);
}
```

Figure 5.2: C program for 20 million summation modified to support compensated summation technique

representable at every iteration) and then makes an attempt to add 1 again, it exceeds the mantissa limit of 23 bits and hence, rounds it to even. Therefore the proceeding iterations painstakingly always produce the same value when adding one repeatedly. The same behaviour can be observed for this program even when adding upto a billion iterations or even more. Single precision commits the same sin again and again.

5.2.1.2 Compensated Summations

One of the most powerful techniques to mitigate the roundoff errors was introduced by Professor William Kahan [23] and has been lately used by many emerging tools like Herbie [38] to help improve the accuracy of programs involving summation over long running loops. This technique was found to work well for floating point numbers since it keep tracks of the errors that accumulate across the loop iterations. When adding a larger number to a smaller number, when the numbers dont overlap against each other, floating point would simply output the larger number as the result, ignoring the bits that dont fit. But compensated summation helps to remember these errors, accumulate them and adds them up to the intermediate sum of every iteration by keeping track of the accumulated error in a separate variable. Although this technique has gained significant attention, it has its own disadvantages and might not work as desired with compiler optimizations [18].

The C code modified to support compensated summation is shown in Figure 5.2.

Table 5.2: Roundoff behaviour of the last 3222786 iterations of the 20 million compensated summation code

Iteration	Floating point mantissa $\times 2^{exponent}$ (The below number is entered to help count digits. The symbol $\downarrow$ is used to mark the maximum limit of mantissa above which the number is usually rounded) `1.12345678901234567890123`$\downarrow$`1234`	Result	Remark
16777215	`1.11111111111111111111111`$\downarrow$ $\times 2^{23}$	16777215	Correct
16777216	`1.00000000000000000000000`$\downarrow$`0`$\times 2^{24}$	16777216	Correct
16777217	`1.00000000000000000000000`$\downarrow$`0`$\times 2^{24}$	16777216	Wrong!
16777218	`1.00000000000000000000001`$\downarrow$`1`$\times 2^{24}$	16777218	Correct
16777219	`1.00000000000000000000010`$\downarrow$`1`$\times 2^{24}$	16777220	Wrong!
⋮	(behaviour repeats as that of iteration 16777217 and 16777218)	⋮	Wrong results equally spaced by 2
19999999	`1.00110001001011010000000`$\downarrow$`1`$\times 2^{24}$	20000000	Wrong!
20000000	`1.00110001001011010000000`$\downarrow$`1`$\times 2^{24}$	20000000	Correct!

The modified program produces a correct final result of 20 million but all the odd numbers starting from 16777216 produce erroneous outputs as can be seen in iterations 16777217, 16777219 and so on until 19999999. While IEEE floats always round to nearest even, it cannot reach the expected result. From a naive computer programmers point of view, compensated summation technique usually mimics twice as many bits as that of the original 32 bit single precision float and should be able to represent the number 20 million exactly with 64 bits. Through compensated summation, since the same program is upgraded to a virtual double precision float for it's intermediate calculations, the program should output 20 million exactly and in fact it does. But the issue here is not lack of precision. Compensated sums work by gradually accumulating the rounding errors that floats introduce to any long running summation. Although this could get rid of the accumulation of rounding errors, this technique is built on top of floats and are still subjected to representation errors like the ones mentioned above. The last 3222786 iterations of the above code are illustrated in Table 5.2.

5.2.1.3 Extended Precision Floating Point and Interval Arithmetic

We modified the 20 million summation problem to support the multi-precision calculations using both MPFR and MPFI libraries that were introduced earlier in Chapter

2. Using MPFR, we used an initial precision of 24 bits and experimented with different rounding modes for the same program. First, we found that changing the rounding modes affect the final answer to a greater extent. The extended precision 'Round to nearest even' behaves the same way as floating point nearest-even rounding mode giving a result of 16777216. But round to $+\infty$ rounds up the result of every iteration after 16777216 thus magnifying the error of the final result to a huge value i.e, 23222784 which is much far from the expected result of 20 million. We also observed that increasing the precision by 1 bit makes the result exactly representable in both round to nearest as well as round to $+\infty$ as can be seen in Table . Thus MPFR relies heavily on the programmer's choice of the bit precision and a poor guesswork could lead to unpredictable results. While extended precision libraries can reduce the occurrence of errors at times, better ways must be found to diagnose the associated numerical malfunctions.

Using the MPFI library, we replaced floats with extended precision interval values [a,b], where a and b are IEEE 16-bit floats. Each endpoint has 3.3 decimal digits of precision and a dynamic range of 1.0×10^{12}. This form of interval arithmetic has for several decades been offering hope for something that would evolve into a sort of automatic error analysis. Using 16-bit MPFI intervals, the expected result falls well within the produced bounds but the bounds were heavily overestimated, a problem which is inherent to any form of interval arithmetic. Although the required precision and exponent could be predicted based on the bounds on the result, this can only be done at the user's own risk.

5.2.1.4 Unum Arithmetic

Type 1 unum arithmetic has adjustable ranges with a maximum of three bits for the length of the exponent field and five for the length of the fraction field. Unums have a precision of up to 9.9 decimals and a dynamic range of 2.4×10^{86}, quite a bit more than offered by 32-bit floats. The number of bits per unum varies from 12 to 49 as needed. Our implementation uses an average of 32 bit operand size after the ubound pairs are converted to their respective single unums after unification. The flexibility of unums comes from the fact that the precision and range of these numbers can be

Table 5.3: Roundoff behaviour of the last 3222786 iterations of the 20 million summation using Unum arithmetic

Iteration	Floating point mantissa $\times 2^{exponent}$ (The below number is entered to help count digits. The symbol ↓ is used to mark the maximum no of fraction bits used) above which the ubit is set to mark inexact result (...) 1.12345678901234567890123↓1234	Result	Remark
16777215	1.11111111111111111111111↓ $\times 2^{23}$	16777215	Correct
16777216	1.00000000000000000000000↓0$\times 2^{24}$	16777216	Correct
16777217	1.00000000000000000000000↓0$\times 2^{24}$	(16777216, 16777218)	bounded correct!
16777218	1.00000000000000000000000 ... 1$\times 2^{24}$	(16777216, 16777220)	bounded correct
⋮	(behaviour repeats as that of iteration 16777217 and 16777218)	⋮	Results equally spaced by (0,2)
19999999	1.01100010010110011111111 ... 1$\times 2^{24}$	(16777216, 23222782)	Correct!
20000000	1.01100010010110011111111 ... 1$\times 2^{24}$	(16777216, 23222784)	Correct!

adjusted (grow or shrink) automatically as and when it is needed. If we cap the range at 36 bits maximum, that usually keeps the average size per unum below 32 bits. Table 5.3 shows the unum arithmetic results obtained for the 20 million summation. Results show that unum arithmetic produces a fairly rigorous bound on the result when compared to interval arithmetic which shoots the upper bound to infinity.

The comparative evaluation of the results for the 20 million summation using various arithmetic methods is listed in Table 5.4. Results show that 32-bit floats fail to produce the correct result for this problem. Extended precision arithmetic occasionally produces the correct result however the precision and rounding modes are usually programmer's guess works. Although compensated sums produce the correct result in this case, this technique couldn't escape the floating point representation errors. In addition to that, the process of rewriting the code to compensate for rounding could be very much error-prone and needs indepth understanding of numerical error analysis. This approach uses much more coding effort from humans and three times as many bit operations for a summation loop. Interval arithmetic and unum arithmetic bounds the

Table 5.4: Arithmetic methods used, results and run time for the 20 million summation using 32-bits maximum vs 32-bits on average

Method	32 bit maxed		32 bit averaged	
	Result	Time(s)	Result	Time(s)
32-bit float	16777216	0.081	-	-
Compensated sums using 32-bit floats	20000000	0.195	-	-
Extended precision,(Round to nearest)	16777216	0.459	2.0e7	0.7498
Extended precision,(Round to $+\infty$)	23222784	0.468	2.0e7	0.7768
Interval arithmetic	[2048, ∞]	0.5488	[32767,∞]	1.2476
Unum arithmetic	(131072, 2.2210e22)	30.702	(16777216,23222784)	27.598

result over intervals. In cases were interval arithmetic produces a pessimistic overestimation of the bound, unum arithmetic produces a fairly rigorous bound but suffers $20\times$ to $50\times$ in performance compared to the interval computations. The time taken to run the 20 million summation code is provided in table 5.4 for two scenarios, a maximum of 32-bits and an average of 32-bits. The reported time is an average measure of running these programs for 10 times. The comparison shows that floats took the lowest amount of time, about one twelfth of a second(reporting a wrong answer in this case) and unums were the most time consuming taking 30.7s to produce a correct bound on the answer.

5.2.2 Non-Dyadic Fractions: Patriot Missile Crash

Dyadic fractions are rational numbers whose denominator is a power of two. These are also the numbers whose binary expansion is finite, for example 1/2 or 3/8 or 1/4. Any non-dyadic fraction such as 1/3 will have an infinite binary representation and for the same reason these numbers are not exactly representable using floats. For instance, 0.1 is a non dyadic fraction.

A long summation of non-dyadic fractions, for instance adding say 0.1 a hundred thousand times can never lead to a correct answer using IEEE 32-bit floats. While the expected answer is *10,000*, IEEE floats give *9998.556641*, an answer off by *1.443359*. For any larger summation, the consequences on the result could be huge. While double

```
float clock_counter () {
int i;
count = 0.0;
for (i = 0; i <(100 * 60 * 60 * 10) ; i++)
  {count = count + 0.1;}
printf ("%f\n", count);
}
```

Figure 5.3: C program for the internal clock counter of the patriot missile

Table 5.5: Arithmetic methods used, results and run time for the Patriot missile simulation using 32-bits maximum vs 32-bits on average

Method	32 bit maxed		Average of 36 bits	
	Result	Time(s)	Result	Time(s)
32-bit float	347024.781250	0.0192	-	-
Compensated sums using 32-bit floats	360000	0.043	-	-
Extended precision,(Round to nearest)	347024.781250	0.026	3.6e6	0.023
Extended precision,(Round to $+\infty$)	3.6e6	0.092	3.6e6	0.143
Interval arithmetic	[1.2800e2,1.3328e106]	0.148	[2.5600e2,1.8358e531]	0.151
Unum arithmetic	(131072, 2.2210e22)	7.445	(340750.0625, 400781.625)	7.714

precision can handle this inaccuracy better than floats, it demands double the amount of bits and resources and proves to be wasteful.

This section examines the case study of the Patriot missile crash recorded on February 25, 1991 during the Gulf war. 28 soldiers were killed and around 100 people were injured due to a simple computer arithmetic error due to the internal clock which calculated time in 10s of seconds. The clock was running since boot (for 100 hours) which calculated time inaccurately. All calculations for the missile used a 24-bit fixed point register. Figure 5.3 shows the code for the internal clock counter of the patriot missile.

5.2.2.1 Comparison Of Floats, MPFR, MPFI and Unums

The expected answer for the code shown in Figure 5.3 is 360000s. But the answer reported by the IEEE 32-bit float is 347024.781250s. For the scud travelling 1.676 m/s, it would've covered a huge distance of approximately 21 km in the elapsed time. This precisely is caused due to the rounding errors accumulated by adding a non-dyadic fraction. Table 5.5 shows the results and run time for the patriot missile clock counter simulation using various arithmetic methods. Compensated summation works correctly in this scenario but imposes atleast three times more resources and hence more burden on the hardware arithmetic. This comes with the exerted revision to the existing code which could prove to be error prone if done carelessly.

Extended precision implementations produce different results with varying rounding modes using the same precision bits. Hence a naive programmer who is prone to alter the rounding modes could easily without any clue end up with a result far from the desired. The missiles internal clock is fairly rigorously bounded with unums with an interval width which is *1.8358e+531* times tighter than bounds produced by the MPFI intervals, but the unum code suffers performance by running approximately 51 times slower.

Chapter 6

Conclusion and Future Work

Arbitrary precision bitwidth analysis proves that bit-level approximations can help reduce hardware resource consumption and yield upto 82% and 66% average reduction in resources when compared to the IEEE standards' double precision and single precision formats respectively. However, the quality-speed tradeoff seems unavoidable in making the right choices for approximation. This demands the need for efficient tools and frameworks to assist in this process to achieve significant cost savings without jeopardising the desired answer quality.

Several alternatives to the standard floating point arithmetic such as extended precision floating point and interval arithmetic reduces the occurrence of errors by increasing the working precision or by presenting hopes to offer a rigorous bound on the answer. But these techniques do not always produce a predictable behaviour accounting for the abnormalities introduced with manual tuning of both the precision and rounding modes. Other techniques such as compensated summation were briefed in this work and our results show that such a technique cannot escape floating point representation errors and rewriting the code is error-prone. In addition to that it would demand a high cost on both resources and the numerical expertise. Unum arithmetic produces a fairly rigorous bound and completely escapes the roundoff errors, yet our results shows that Type 1 unums suffer from performance and could be as slow as $20\times$ to $50\times$ compared to interval arithmetic.

We also emphasize that the above mentioned techniques need to address some of the most important challenges as follows,

- Differentiate the rounding errors from coding errors
- Desired accuracy with limited precision or storage requirements
- Automatic error analysis with less numerical expertise
- Higher programmer productivity
- Higher performance on bandwidth limited codes

6.1 Future Work

We extend this work by exploring the scope of various arithmetic techniques on global optimization problems. As an alternative to the floating point, we make use of unum arithmetic to study the constraint solving and optimization problems. We implemented the first phase of a unum constraint solver which can solve a system of linear equations with integer and floating point constraints, more efficiently than a brute force search implemented using floats. Appendix B lists the code for the Unum solver. The solver supports automatic tuning of precision and range starting from the minimum exponent and fraction size length of 1. Other rigorous applications of constrain solvers such as robot arm inverse kinematics has been tried out by the solver. In the preliminary study, one particular concern is that the unum solver produces several variants to the solution set as that of the one reported by the existing interval solvers. This could be a result of the splitting strategy used which in this case is a breadth first search and the inadequacy in estimating the desired number of precision bits for each split. A possible direction is to identify a split-prune strategy that best fits a specific application and helps identify a unique set of solutions if that exist and if not provide a clue of whether a solution exists or not. We also found that providing a reasonable choice of the unum environment, i.e, the precision and range from the very beginning of the search can help reduce the search time to improve performance.

Bibliography

[1] Tor M Aamodt and Paul Chow. Compile-time and instruction-set methods for improving floating-to fixed-point conversion accuracy. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):26, 2008.

[2] Douglas N Arnold. The patriot missile failure. *Retrieved October*, 3:2010, 2000.

[3] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[4] Subhankar Bhattacharjee, Sanjib Sil, and Amlan Chakrabarti. Evaluation of power efficient fir filter for fpga based dsp applications. *Procedia Technology*, 10:856–865, 2013.

[5] 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.

[6] Vinay Kumar Chippa, Debabrata Mohapatra, Kaushik Roy, Srimat T. Chakradhar, and Anand Raghunathan. Scalable effort hardware design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(9):2004–2016, 2014.

[7] Jaeyong Chung and Lok-Won Kim. Bit-width optimization by divide-and-conquer for fixed-point digital signal processing systems. *Computers, IEEE Transactions on Digital Signal Processing Systems*, pages 3091–3101, 2015.

[8] Jason Cong, Yiping Fan, Guoling Han, Yizhou Lin, Junjuan Xu, Zhiru Zhang, and Xu Cheng. Bitwidth-aware scheduling and binding in high-

level synthesis. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 856–861. ACM, 2005.

[9] Jason Cong, Karthik Gururaj, Bin Liu, Chunyue Liu, Zhiru Zhang, Sheng Zhou, and Yi Zou. Evaluation of static analysis techniques for fixed-point precision optimization. In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 231–234. IEEE, 2009.

[10] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011.

[11] Luiz Henrique de Figueiredo and Jorge Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1):147–158, 2004.

[12] Yang Ding and Weng Fai Wong. Bit-width analysis for general applications, http://hdl.handle.net/1721.1/7412,. 2005.

[13] Tom Feist. Vivado design suite. *White Paper*, 5, 2012.

[14] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[15] A. A. Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. In *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pages 158–165, 2002.

[16] 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.

[17] Andreas Gerstlauer, Rainer Dmer, Junyu Peng, and Daniel D Gajski. *System design: a practical guide with SpecC*. Springer Science & Business Media, 2012.

[18] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[19] Thorsten Grtker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Springer Science & Business Media, 2002.

[20] John L Gustafson. *The End of Error: Unum Computing*, volume 24. CRC Press, 2015.

[21] Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. Efficient floating point precision tuning for approximate computing. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 63–68. IEEE, 2017.

[22] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. 2009.

[23] William Kahan. Kahan compensated summation algorithm.

[24] Kum Ki-Il, Kang Jiyang, and Sung Wonyong. A floating-point to integer c converter with shift reduction for fixed-point digital signal processors. In *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*, volume 4, pages 2163–2166 vol.4, 1999.

[25] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-point optimization utility for c and c++ based digital signal processing programs. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 45(11):1455–1464, 1998.

[26] Adam Bruce Kinsman and Nicola Nicolici. Bit-width allocation for scientific computations, 2014.

[27] Michael O Lam, Jeffrey K Hollingsworth, Bronis R de Supinski, and Matthew P LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 369–378. ACM, 2013.

[28] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-guaranteed bit-width optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(10):1990–2000, 2006.

[29] 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.

[30] Emin Martinian. Fast fourier transform.

[31] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. Chisel: reliability-and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 309–328. ACM, 2014.

[32] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12, 2008.

[33] Ramon E. Moore. *Interval analysis*. Prentice-Hall, 1966.

[34] J-P Moreau. Function approximations in c/c++. 2014.

[35] Nedialko S. Nedialkov, Vladik Kreinovich, and Scott A. Starks. Interval arithmetic, affine arithmetic, taylor series methods: Why, what next? *Numerical Algorithms*, 37(1):325–336, 2004.

[36] Peter G Neumann. Risks to the public in computers and related systems. *ACM SIGSOFT Software Engineering Notes*, 26(6):6–15, 2001.

[37] William G Osborne, Ray CC Cheung, J 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.

[38] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Notices*, 50(6):1–11, 2015.

[39] Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the mpfi library. *Reliable computing*, 11(4):275–290, 2005.

[40] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334. ACM, 2006.

[41] Martin C Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *ACM SIGPLAN Notices*, volume 42, pages 369–386. ACM, 2007.

[42] Eric Roberts. Stanford data compression notes: Dct equation.

[43] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H Bailey, and David Hough. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1074–1085. ACM, 2016.

[44] Cindy Rubio-Gonzlez, Cuong Nguyen, Hong Nguyen, James Demmel, William Kahan, Koushik Sen, David Bailey, Costin Iancu, and David Hough. Precimonious: tuning assistant for floating-point precision. pages 1–12. ACM.

[45] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[46] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134. ACM, 2011.

[47] Diego F Snchez, Daniel M Muoz, Carlos H Llanos, and Mauricio Ayala-Rincn. Parameterizable floating-point library for arithmetic operations in fpgas. In *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, page 40. ACM, 2009.

[48] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *ACM SIGPLAN Notices*, volume 35, pages 108–120. ACM, 2000.

[49] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[50] MPFR TEAM et al. The mpfr library: Algorithms and proofs. 2012.

[51] Jonathan Ying Fai Tong, David Nagle, and Rob Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(3):273–286, 2000.

[52] Sattar Vakili, JM Langlois, and Guy Bois. Enhanced precision analysis for accuracy-aware bit-width optimization using affine arithmetic. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(12):1853–1865, 2013.

[53] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: a modeling language for global optimization*. MIT press, 1997.

[54] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Computing approximately, and efficiently. pages 748–751. EDAA.

[55] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *Proceedings of the 52nd Annual Design Automation Conference*, page 120. ACM, 2015.

[56] Bin Wu. *Dynamic range estimation and bitwidth determination*. PhD thesis, ProQuest Dissertations Publishing, 2006.

[57] Linsheng Zhang, Yan Zhang, and Wenbiao Zhou. Floating-point to fixed-point transformation using extreme value theory. In *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*, pages 271–276. IEEE, 2009.

Appendix A

An Example of Float-to-Fixed Point Converted Code

A.1 Strassen's Matrix Multiplication

ORIGINAL C CODE

```
#include"mat_mul.h"
#include<math.h>
#include<limits.h>

// Original floating point code
void mat(float x[M][N],float y[M][N])
{
        int i,j;
    p0 = (x[0][0] + x[1][1]) * (y[0][0]+ y[1][1]);
    p1 = (x[1][0] + x[1][1]) * y[0][0];
    p2 = x[0][0] * (y[0][1] - y[1][1]);
    p3 = x[1][1] * (y[1][0] - y[0][0]);
    p4 = (x[0][0] + x[0][1]) * y[1][1];
    p5 = (x[1][0] - x[0][0]) * (y[0][0] + y[0][1]);
    p6 = (x[0][1] - x[1][1]) * (y[1][0] + y[1][1]);
```

```
    z[0][0] = p0 + p3- p4 + p6;
    z[0][1] = p2 + p4;
    z[1][0] = p1 + p3;
    z[1][1] = p0 + p2 - p1 + p5;

   for (i=0;i<M; i++)
        for(j=0;j<N;j++)
         printf("\n Original Matrix Output z[%d][%d] = %f \n "
         ,i,j,z[i][j]);
}
```

Header file:

```
#ifndef MAT_MUL_H_
#define MAT_MUL_H_
#include<math.h>
//Rows and columns of the matrix
#define M 2
#define N 2

        float p0,p1,p2,p3,p4,p5,p6;
        float z[M][N];
        void mat(float x[M][N],float y[M][N]);
        float a[M][N], b[M][N];
#endif
```

FIXED-POINT CODE AFTER CONVERSION (WITH SQNR ERROR CALCULATION)

```
#include<stdlib.h>
#include<time.h>
#include"mat_mul.h"
float z[M][N];
```

```
ap_fixed <16,5,AP_RND, AP_SAT> z1[M][N];
int main()
{       int i,j,k;
        float sqnr_avg;
        float error_avg;
        float sqnr_sum = 0.0;
        float error_sum = 0.0;
        float a[M][N], b[M][N];
        float rel_error_avg = 0.0;
#ifdef ERROR_CALC
    float value,sqnr,error;
    float numer = 0.0;
    float denom = 0.0;
    float abs_error;
    float rel_error;
    float sqr_abs_error;
    float sqr_signal_out;
    float rel_error_sum = 0.0;
    float rel_error_avg_sum = 0.0;
    float rel_error_avg_sum_avg = 0.0;
#endif
        for(k=1;k<=10;k++){
         srand(1);
      printf("\n Generating inputs for
      training set: %d\n",k);
      for (i=0;i<M;i++)
      for (j=0;j<N;j++){
           a[i][j]= (float)rand()/RANDMAX;
           printf("\n Matrix a[%d][%d] = %f  ",i,j,a[i][j]);
           }
         printf("\n");
           for(i=0;i<M;i++)
```

```
            for(j=0;j<N;j++) {
                  b[i][j]= (float)rand()/RAND_MAX;
                  printf("\n Matrix b[%d][%d] = %f  ",i,j,b[i][j]);
                  }
                  printf("\n\n Running software simulations for
                  training set: %d",k);
#ifdef FIXED_POINT
                  ap_fixed<12,2,AP_RND, AP_SAT>  a_new[M][N];
                  ap_fixed<12,2,AP_RND, AP_SAT>  b_new[M][N];
            for (i=0;i<M; i++)
            for(j=0;j<N;j++)
                  a_new[i][j]= a[i][j];
            for (i=0;i<M; i++)
            for(j=0;j<N;j++)
                  b_new[i][j]=  b[i][j];
#endif
#ifdef FLOATING_POINT
                  mat(a,b);
#endif
#ifdef FIXED_POINT
                  mat_conv(a_new,b_new);
#endif
//calculating SQNR
#ifdef ERROR_CALC
          for (i=0;i<M; i++)
          for(j=0;j<N;j++){
       abs_error = fabs(z[i][j] - (float) z1[i][j]);
            rel_error = fabs(abs_error/(float) z1[i][j]);
            rel_error_sum = rel_error_sum + rel_error;
            printf(" \n Expected output @ %d: %f \t\t
            Hardware_simulation output @ %d: %f \t\t
            abs_error = %f",i,z[i][j],i,(float)z1[i][j],abs_error);
```

```
            sqr_abs_error = abs_error*abs_error;
            sqr_signal_out = (z[i][j]) * (z[i][j]);
            numer = numer + sqr_signal_out;
            denom = denom + sqr_abs_error;
            }
            rel_error_avg = rel_error_sum/k;
            rel_error_avg_sum = rel_error_avg_sum + rel_error_avg;
            rel_error_avg_sum_avg = rel_error_avg_sum_avg
                                  + rel_error_avg_sum;
            value = numer/denom;
            sqnr = 10 * log10(value);
            sqnr_sum = sqnr_sum +sqnr;
            error = 1.0/value;
            error_sum = error_sum + error;
            printf("\n sqnr in db at simulation %d= %lf",k,sqnr);
            printf("\n sqnr_error_rate at simulation %d= %lf",
            k, error);
            sqnr_avg = sqnr_sum/k;
            error_avg = error_sum/k;
            //Average SQNR of j simulations
            printf("\n Avg_sqnr = %.5lf",sqnr_avg);
             }
#endif
        return 0;
}
```

Fixed-point Function:

```
void mat_conv(ap_fixed<12,2,AP_RND, AP_SAT> x[M][N],
ap_fixed<12,2, AP_RND, AP_SAT> y[M][N])
{
        int i,j;
        extern ap_fixed<16,5, AP_RND, AP_SAT> z1[M][N];
```

```
    ap_fixed <15,5, AP_RND, AP_SAT> p0;
    ap_fixed <16,5, AP_RND, AP_SAT> p1;
    ap_fixed <16,5, AP_RND, AP_SAT> p2;
    ap_fixed <15,5, AP_RND, AP_SAT> p3;
    ap_fixed <15,5, AP_RND, AP_SAT> p4;
    ap_fixed <15,5, AP_RND, AP_SAT> p5;
    ap_fixed <15,5, AP_RND, AP_SAT> p6;

    p0 = (x[0][0] + x[1][1]) * (y[0][0]+ y[1][1]);
    p1 = (x[1][0] + x[1][1]) * y[0][0];
    p2 = x[0][0] * (y[0][1] - y[1][1]);
    p3 = x[1][1] * (y[1][0] - y[0][0]);
    p4 = (x[0][0] + x[0][1]) * y[1][1];
    p5 = (x[1][0] - x[0][0]) * (y[0][0] + y[0][1]);
    p6 = (x[0][1] - x[1][1]) * (y[1][0] + y[1][1]);

    z1[0][0] = p0 + p3- p4 + p6;
    z1[0][1] = p2 + p4;
    z1[1][0] = p1 + p3;
    z1[1][1] = p0 + p2 - p1 + p5;

   for (i=0;i<M; i++)
   for(j=0;j<N;j++)
    printf("\n reduced Matrix Output z[%d][%d] = %f \n ",
    i,j,(float)z1[i][j]);
}
```

Appendix B

Unum Constraint Solver Code

B.1 Solver for Robot Arm Inverse Kinematics

The Unum solver makes use of the C unum arithmetic functions like addition, subtraction, square-root and multiplication which are implemented by our group. The inverse kinematics equations have been chosen from the book Numerica [53]

CONSTRAINT CHECKER FOR KINEMATICS EQUATIONS

```
bool checkConstraintRobotarm(ubound_t s1,ubound_t s2,
ubound_t s3,ubound_t s4, ubound_t s5, ubound_t s6,
ubound_t c1, ubound_t c2, ubound_t c3, ubound_t c4,
ubound_t c5, ubound_t c6,bool* pass, bool* noInf)
{
 bool left_open[12],right_open[12],conValid[12],
 conPass[12];
 ubound_t con[12],t,u,THREE,TWO,THREE_c2,TWO_c3,
 c5_s6,u_c6,s1_c5, THREE_s2,TWO_s3,s1_SQUARE,s2_SQUARE,
 s3_SQUARE,s4_SQUARE, s5_SQUARE,s6_SQUARE,c1_SQUARE,
 c2_SQUARE,c3_SQUARE,c4_SQUARE, c5_SQUARE,c6_SQUARE;
 double left[12],right[12],CON_RHS[12]; int i;
```

```
*pass = false;
*noInf = true;
x2u(3,&THREE.left_bound);
x2u(3,&THREE.right_bound);
x2u(2,&TWO.left_bound);
x2u(2,&TWO.right_bound);
//t = 3c2 + 2c3 + c4
timesubound(&THREE_c2,THREE,c2);
timesubound(&TWO_c3,TWO,c3);
plusubound(&t,THREE_c2,TWO_c3);
plusubound(&t,t,c4);
//u = c2 + c3 + c4
plusubound(&u,c2,c3);
plusubound(&u,u,c4);
CON_RHS[0] = 0.4077;
CON_RHS[1] = 1.9115;
CON_RHS[2] = 1.9791;
CON_RHS[3] = 4.0616;
CON_RHS[4] = 1.7172;
CON_RHS[5] = 3.9701;
CON_RHS[6] = 1;
CON_RHS[7] = 1;
CON_RHS[8] = 1;
CON_RHS[9] = 1;
CON_RHS[10] = 1;
CON_RHS[11] = 1;
//con1 = (s2 - s3 - s4) * (c5*s6) + (u * c6)
minusubound(&con[0],s2,s3);
minusubound(&con[0],con[0],s4);
timesubound(&c5_s6,c5,s6);
timesubound(&con[0],con[0],c5_s6);
timesubound(&u_c6,u,c6);
```

```
 plusubound(&con[0],con[0],u_c6);
//con2 = (c1*u*s5)+s1*c5
 timesubound(&con[1],c1,u);
 timesubound(&con[1],con[1],s5);
 timesubound(&s1_c5,s1,c5);
 plusubound(&con[1],con[1],s1_c5);
//con3 = (s2 + s3 + s4) * s5
 plusubound(&con[2], s2, s3);
 plusubound(&con[2],con[2],s4);
 timesubound(&con[2],con[2],s5);
//con4 = c1*t
 timesubound(&con[3],c1,t);
//con5 = s1*t
 timesubound(&con[4],s1,t);
//con6 = 3s2 + 2s3 + s4
 timesubound(&THREE_s2, THREE, s2);
 timesubound(&TWO_s3, TWO, s3);
 plusubound(&con[5], THREE_s2, TWO_s3);
 plusubound(&con[5], con[5], s4);
 squareubound(&s1_SQUARE, &s1);
 squareubound(&c1_SQUARE,&c1);
 plusubound(&con[6], s1_SQUARE,c1_SQUARE);
 squareubound(&s2_SQUARE, &s2);
 squareubound(&c2_SQUARE,&c2);
 plusubound(&con[7], s2_SQUARE,c2_SQUARE);
 squareubound(&s3_SQUARE, &s3);
 squareubound(&c3_SQUARE,&c3);
 plusubound(&con[8], s3_SQUARE,c3_SQUARE);
 squareubound(&s4_SQUARE, &s4);
 squareubound(&c4_SQUARE,&c4);
 plusubound(&con[9], s4_SQUARE,c4_SQUARE);
 squareubound(&s5_SQUARE, &s5);
```

```
squareubound(&c5_SQUARE,&c5);
plusubound(&con[10], s5_SQUARE,c5_SQUARE);
squareubound(&s6_SQUARE, &s6);
squareubound(&c6_SQUARE,&c6);
plusubound(&con[11], s6_SQUARE,c6_SQUARE);
//commenting out the ubound answers
  #ifdef PRINT_SPLITS
    for(i=0;i<=11;i++){
     printf("\n ubound answer for eqtn %d:",i);
     ub2f_view(&con[i],&left[i],&right[i],&left_open[i],
    &right_open[i]);
     printf("\t (%lf)",CON_RHS[i]);
     }
   #endif
     for(i=0;i<=11;i++){
       ub2f(&con[i],&left[i],&right[i],&left_open[i],
      &right_open[i]);
       conValid[i] = 0;
       conPass[i] = 0;
       }
     for(i=0;i<=11;i++){
      if(isinf(left[i]) || isinf(right[i])){
         *noInf = false;
          break;
         }
       }
      for(i=0;i<=11;i++){
//If both are closed
   if(!left_open[i] && !right_open[i]){
     if((CON_RHS[i] >= left[i]) && (CON_RHS[i] <= right[i]))
         conValid[i] = 1;
     if((CON_RHS[i] == left[i]) && (CON_RHS[i] == right[i])){
```

```
        conPass[i] = 1;
          }
        }
  if(left_open[i] && right_open[i])
    if((CON_RHS[i] >left[i]) && (CON_RHS[i] <right[i]))
      conValid[i] = 1;;

  if(!left_open[i] && right_open[i])
    if((CON_RHS[i] >=left[i]) && (CON_RHS[i] <right[i]))
      conValid[i] = 1;;

  if(left_open[i] && !right_open[i])
    if((CON_RHS[i] >left[i]) && (CON_RHS[i] <=right[i]))
      conValid[i] = 1;;
}

  if((conValid[0] && conValid[1] && conValid[2]
  && conValid[3]  && conValid[4] && conValid[5]
  && conValid[6] && conValid[7]  && conValid[8]
  && conValid[9] && conValid[10] && conValid[11]))
    {
      #ifdef PRINT_SPLITS
      printf("\n All constraints are satisfied
      for the above combination");
      #endif
      if((conPass[0] && conPass[1] && conPass[2]
      && conPass[3]  && conPass[4] && conPass[5]
      && conPass[6] && conPass[7]  && conPass[8]
      && conPass[9] && conPass[10] && conPass[11]))
        *pass = true;
     return true;
   }
```

```
else{
   #ifdef PRINT_SPLITS
   printf("\n Constraints not satisfied
   for the above combination");
   #endif
    return false;
    }
}
```

UNUM SOLVER

```
int main(int argc, char *argv[]) {
double lb,rb,ub1_lb,ub1_rb,lb_s[6],rb_s[6],lb_c[6],
rb_c[6],lb_greater,rb_greater,width[12],left[12],right[12],
lb_san[6],rb_san[6],lb_s1,rb_s1,lb_s2,rb_s2,lb_s3,rb_s3,
lb_s4,rb_s4,lb_s5,rb_s5,lb_s6,rb_s6,lb_c1,rb_c1,lb_c2,
rb_c2,lb_c3,rb_c3,lb_c4,rb_c4,lb_c5,rb_c5,lb_c6,rb_c6,
abs_width_fs[6],abs_width_fc[6],tot_abs_width_s=0.0,
tot_abs_width_c=0.0,tot_abs_width,avg_abs_width,
CURR_MIN_ULP=0.25,PREV_MIN_ULP=1;
bool l_open,r_open,left_open[12], right_open[12],
lopen_san[6],ropen_san[6],ub1_lopen,ub1_ropen,
greater_lopen,greater_ropen,sl_open[6],sr_open[6],
cl_open[6],cr_open[6],xl_open,xr_open,yl_open,
yr_open,zl_open,zr_open,valid_comb=0, pass=false,
noInf=true,s1l_open,s2l_open,s3l_open,s4l_open,
s5l_open,s6l_open,s1r_open,s2r_open,s3r_open,
s4r_open,s5r_open,s6r_open,c1l_open,c2l_open,
c3l_open,c4l_open,c5l_open,c6l_open,c1r_open,
c2r_open,c3r_open,c4r_open,c5r_open,c6r_open,
s1_NO_SPLIT=0, s2_NO_SPLIT=0, s3_NO_SPLIT=0,
s4_NO_SPLIT=0, s5_NO_SPLIT=0, s6_NO_SPLIT=0,
```

```
c1_NO_SPLIT=0, c2_NO_SPLIT=0, c3_NO_SPLIT=0,
c4_NO_SPLIT=0, c5_NO_SPLIT=0, c6_NO_SPLIT=0,valid_cnt=0;
ubound_t initial_ub ,ub1,ub2,ub3,ub4,ub5,ub6,ub7,ub8,
ub9,ub10,ub11,ub12,left_ub ,right_ub ,s_SQUARE[6],
c_SQUARE[6],s[6],c[6],s_SQUARE_plus_c_SQUARE[6],
ubound_ranges[3],greater_ubound ,s1_split[3],
s2_split[3],s3_split[3],s4_split[3],s5_split[3],
s6_split[3],c1_split[3],c2_split[3],c3_split[3],
c4_split[3],c5_split[3],c6_split[3],sols_s1[10],
sols_s2[10],sols_s3[10],sols_s4[10],sols_s5[10],
sols_s6[10],sols_c1[10],sols_c2[10],sols_c3[10],
sols_c4[10],sols_c5[10],sols_c6[10],x[10],temp,
ub_fss[6],ub_fcc[6],si[6],ci[6];
gbound_t gb1,gb2,gb3,gb4,gb5,gb6,gb7,gb8,gb9,gb10,
gb11,gb12,gb_fss[6],gb_fcc[6],gbound_ranges[3];
ubFloat_t fs[6],fc[6],fss[6],fcc[6];
int i,j=0,k,number_of_ranges, passed_cnt=0,sols_cnt=0,
fn_eval=0,exp=2,frac=3,exp_cnt,frac_cnt,queue_cnt=0;
Queue *q = createQueue(12);
FQueue *r = createFQueue(12);
//Set the initial intervals
        l_open = 1;
        lb =  -100000000;
        rb =   100000000;
        r_open = 1;
//Set the environment
   set_env(exp,frac);
//convert float intervals into ubounds and push it into
 queue q if its a valid combination
    x2u(lb,&gb1.left_bound);
    x2u(rb,&gb1.right_bound);
    gb1.left_open = l_open;
```

```
    gb1.right_open = r_open;
//Get the ubounds from gbounds
    get_ubound_result_from_gbound(&ub1, &gb1);
    uboundview(ub1);
    printf("\n Starting search with (-1,1) in a (%d,%d)
    environment",exp,frac);
    valid_comb = checkConstraintRobotarm(ub1,ub1,ub1,ub1,
    ub1,ub1,    ub1,ub1,ub1,ub1,ub1,ub1,&pass,&noInf);
    fn_eval++;
    if(valid_comb) {
    for(i=0;i<=11;i++)
    push(q,ub1);
    }
REPEATPRUNING:
//If queue is not empty, dequeue the first valid split
and split the greater ubound further into three.
Check which splits satisfy all the constraints and enqueue them
while(!isEmpty(q)){
  queue_cnt++;
  for(i=0;i<=5;i++) {
  s[i] = pop(q);
  ub2f(&s[i],&lb_s[i],&rb_s[i],&sl_open[i],&sr_open[i]);
  }
 for(i=0;i<=5;i++) {
        c[i] = pop(q);
        ub2f(&c[i],&lb_c[i],&rb_c[i],&cl_open[i],&cr_open[i]);
        }
        printf("\n");
        for(i=0;i<=5;i++){
        printf("\t s%d:",i);
        ub2f_view(&s[i],&lb_s[i],&rb_s[i],&sl_open[i],
        &sr_open[i]);
```

```
          printf("\t\t ");
          }
          printf("\n");
          for(i=0;i<=5;i++){
          printf("\t c%d:",i);
          ub2f_view(&c[i],&lb_c[i],&rb_c[i],&cl_open[i],
          &cr_open[i]);
          printf("\t\t ");
          }
/**SANITY CHECK: for equations 7 to 12
//is si^2 + ci^2 == 1?*/
for(i=0;i<=5;i++){
squareubound(&s_SQUARE[i], &s[i]);
squareubound(&c_SQUARE[i],&c[i]);
plusubound(&s_SQUARE_plus_c_SQUARE[i],
s_SQUARE[i],c_SQUARE[i]);
//printf("\n s%d^2+c%d^2=",i,i);
//ub2f_view(&s_SQUARE_plus_c_SQUARE[i],&lb_san[i],
&rb_san[i],&lopen_san[i],&ropen_san[i]);
ub2f(&s_SQUARE_plus_c_SQUARE[i],&lb_san[i],
&rb_san[i],&lopen_san[i],&ropen_san[i]);
if(!lopen_san[i] && !ropen_san[i]){
  if((1 >= lb_san[i]) && (1 <= rb_san[i])==0)
    printf("\n SANITY CHECK FAILED");
    if(((1 == lb_san[i]) && (1 == rb_san[i]))==0)
    printf("\n SANITY CHECK FAILED");
          }
  if(lopen_san[i] && ropen_san[i]){
    if(((1 > lb_san[i]) && (1 < rb_san[i]))==0)
          printf("\n SANITY CHECK FAILED");
            }
```

```
  if(!lopen_san[i] && ropen_san[i]){
    if(((1 >= lb_san[i]) && (1 < rb_san[i]))==0)
         printf("\n SANITY CHECK FAILED");
                  }

  if(lopen_san[i] && !ropen_san[i]){
    if(((1 > lb_san[i]) && (1 <= rb_san[i]))==0)
         printf("\n SANITY CHECK FAILED");
          }
    }
number_of_ranges = 0;
//check which of the twelve splits are bigger
to split this ubound further
for(i=0;i<=5;i++)
x[i] = s[i];
for(i=0,j=6;i<=5 && j<=11;i++,j++){
x[j] = c[i];
}
//set the initial ubound as the greatest
greater_ubound = x[0];
ub2f(&greater_ubound,&lb_greater,&rb_greater,
&greater_lopen,&greater_ropen);
for(i=0;i<=11;i++)
  ub2f(&x[i],&left[i],&right[i],&left_open[i],
 &right_open[i]);
//find the greater ubound
for(i=0;i<=11;i++){
  width[i] = fabs(right[i] - left[i]);
  if((width[i]) > fabs(rb_greater - lb_greater)){
          greater_ubound = x[i];
          ub2f(&greater_ubound,&lb_greater,&rb_greater,
         &greater_lopen,&greater_ropen);
```

```
        }
 }
//sorting the elements in descending order
for(i=0;i<11;i++)
{
 for(j=i+1;j<=11;j++)
  {
   ub2f(&x[i],&left[i],&right[i],
   &left_open[i],&right_open[i]);
   ub2f(&x[j],&left[j],&right[j],
   &left_open[j],&right_open[j]);

   if((right[i] - left[i]) < (right[j] - left[j])){
        temp = x[i];
        x[i] = x[j];
        x[j] = temp;
            }
          }
   }
 //If the left bound is -infinity and the
 right bound is +infinity,
 then set this bound as the greatest bound
 for(i=0;i<=5;i++){
  if((lb_s[i] == -INFINITY) && (rb_s[i] == INFINITY))
           greater_ubound = s[i];
        }

  for(i=0;i<=5;i++){
  if((lb_c[i] == -INFINITY) && (rb_c[i] == INFINITY))
           greater_ubound = c[i];
        }
```

```
for(j=0;j<=11;j++){
//If the number of ranges is 0, and the last ubound
is not yet reached,then continue splitting
 the next ubound
if((number_of_ranges == 0)){
          ub2f(&greater_ubound,&lb_greater,&rb_greater,
          &greater_lopen,&greater_ropen);
          number_of_ranges = splitub(lb_greater,rb_greater,
          greater_lopen,greater_ropen,greater_ubound,
          gbound_ranges,ubound_ranges);
         if((number_of_ranges == 0)&& j==0)
            s1_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==1)
            s2_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==2)
            s3_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==3)
            s4_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==4)
            s5_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==5)
            s6_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==6)
            c1_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==7)
            c2_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==8)
            c3_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==9)
            c4_NO_SPLIT = 1;
         else if((number_of_ranges == 0)&& j==10)
            c5_NO_SPLIT = 1;
```

```
        else{
          if((number_of_ranges == 0)&& j==11)
           c6_NO_SPLIT = 1;
                }
    }
    if(number_of_ranges>0)
          break;
    }
//If the number of ranges is greater than zero,
then split into 3 and generate 3 different trials
for that split and continue splitting with
the next dimension
if(number_of_ranges>0){
 for(i=0; i<number_of_ranges; i++){

   //check if all 3 splits satisfy the constraints.
//If so, don't enqueue any of them. No information is
obtained by splitting that trial
 //get_ubound_result_from_gbound(&ubound_ranges[i],
 &gbound_ranges[i]);
 ubound_ranges[i].uflag = gbound_ranges[i].left_open;
 ubound_ranges[i].vflag = gbound_ranges[i].right_open;
 //printf("\n Splitting the greater bound");
 //If s1 is greater
if(unum_compare(greater_ubound.left_bound,
s[0].left_bound)&&unum_compare(greater_ubound.right_bound,
s[0].right_bound) )
{
 printf("\n \nSplitting s1\n");
 s1_split[i] = ubound_ranges[i];
 s2_split[i] = s[1];
 s3_split[i] = s[2];
```

```
  s4_split[i] = s[3];
  s5_split[i] = s[4];
  s6_split[i] = s[5];

  c1_split[i] = c[0];
  c2_split[i] = c[1];
  c3_split[i] = c[2];
  c4_split[i] = c[3];
  c5_split[i] = c[4];
  c6_split[i] = c[5];
#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
```

```
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
}
//If s2 is greater
else if(unum_compare(greater_ubound.left_bound,
s[1].left_bound)&&unum_compare(greater_ubound.right_bound,
s[1].right_bound) )
 {
  printf(" \n\nSplitting s2\n");
 s1_split[i] = s[0];
 s2_split[i] = ubound_ranges[i];
 s3_split[i] = s[2];
 s4_split[i] = s[3];
 s5_split[i] = s[4];
 s6_split[i] = s[5];

 c1_split[i] = c[0];
 c2_split[i] = c[1];
 c3_split[i] = c[2];
 c4_split[i] = c[3];
 c5_split[i] = c[4];
 c6_split[i] = c[5];
#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
```

```
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
}

//If s3 is greater
else if(unum_compare(greater_ubound.left_bound,s[2].left_bound)&&
unum_compare(greater_ubound.right_bound,s[2].right_bound) ) {
 // printf(" \n\nSplitting s3\n");

 s1_split[i] = s[0];
 s2_split[i] = s[1];
 s3_split[i] = ubound_ranges[i];
 s4_split[i] = s[3];
 s5_split[i] = s[4];
 s6_split[i] = s[5];
```

```
 c1_split[i] = c[0];
 c2_split[i] = c[1];
 c3_split[i] = c[2];
 c4_split[i] = c[3];
 c5_split[i] = c[4];
 c6_split[i] = c[5];
#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
```

```
}
//If s4 is greater
else if(unum_compare(greater_ubound.left_bound,s[3].left_bound)&&
unum_compare(greater_ubound.right_bound,s[3].right_bound) ){
        // printf(" \n\nSplitting s4\n");

         s1_split[i] = s[0];
         s2_split[i] = s[1];
         s3_split[i] = s[2];
         s4_split[i] = ubound_ranges[i];
         s5_split[i] = s[4];
         s6_split[i] = s[5];

         c1_split[i] = c[0];
         c2_split[i] = c[1];
         c3_split[i] = c[2];
         c4_split[i] = c[3];
         c5_split[i] = c[4];
         c6_split[i] = c[5];



ubound_ranges[i];
#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
```

```
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
                    }
//If s5 is greater
else if(unum_compare(greater_ubound.left_bound,s[4].left_bound)
&& unum_compare(greater_ubound.right_bound,s[4].right_bound) ){
// printf(" \n\nSplitting s5\n");

 s1_split[i] = s[0];
 s2_split[i] = s[1];
 s3_split[i] = s[2];
 s4_split[i] = s[3];
 s5_split[i] = ubound_ranges[i];
 s6_split[i] = s[5];

 c1_split[i] = c[0];
 c2_split[i] = c[1];
 c3_split[i] = c[2];
```

```
 c4_split[i] = c[3];
 c5_split[i] = c[4];
 c6_split[i] = c[5];
#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
}
//If s6 is greater
else if(unum_compare(greater_ubound.left_bound,
```

```
s[5].left_bound)&& unum_compare(greater_ubound.right_bound,
s[5].right_bound) ){
// printf(" \n\nSplitting s6\n");

 s1_split[i] = s[0];
 s2_split[i] = s[1];
 s3_split[i] = s[2];
 s4_split[i] = s[3];
 s5_split[i] = s[4];
 s6_split[i] = ubound_ranges[i];

 c1_split[i] = c[0];
 c2_split[i] = c[1];
 c3_split[i] = c[2];
 c4_split[i] = c[3];
 c5_split[i] = c[4];
 c6_split[i] = c[5];
#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
```

```
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
                }
//If c1 is greater
else if(unum_compare(greater_ubound.left_bound,c[0].left_bound)&&
unum_compare(greater_ubound.right_bound,c[0].right_bound) ){
                //  printf(" \n\nSplitting c1\n");

 s1_split[i] = s[0];
 s2_split[i] = s[1];
 s3_split[i] = s[2];
 s4_split[i] = s[3];
 s5_split[i] = s[4];
 s6_split[i] = s[5];

 c1_split[i] = ubound_ranges[i];
 c2_split[i] = c[1];
 c3_split[i] = c[2];
 c4_split[i] = c[3];
 c5_split[i] = c[4];
 c6_split[i] = c[5];
#ifdef PRINT_SPLITS
```

```
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
                    }

//If c2 is greater
else if(unum_compare(greater_ubound.left_bound,c[1].left_bound)&&
unum_compare(greater_ubound.right_bound,c[1].right_bound) ){
//   printf("\n\nSplitting c2\n");
```

```
  s1_split[i] = s[0];
  s2_split[i] = s[1];
  s3_split[i] = s[2];
  s4_split[i] = s[3];
  s5_split[i] = s[4];
  s6_split[i] = s[5];

  c1_split[i] = c[0];
  c2_split[i] = ubound_ranges[i];
  c3_split[i] = c[2];
  c4_split[i] = c[3];
  c5_split[i] = c[4];
  c6_split[i] = c[5];

#ifdef PRINT_SPLITS
 printf("\n s1 split:");
 ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
 printf("\t s2 split:");
 ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
 printf("\t s3 split:");
 ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
 printf("\t s4 split:");
 ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
 printf("\t s5 split:");
 ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
 printf("\t s6 split:");
 ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
 printf("\n c1 split:");
 ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
 printf("\t c2 split:");
 ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
 printf("\t c3 split:");
```

```
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
              }
//If c3 is greater
else if(unum_compare(greater_ubound.left_bound,c[2].left_bound)&&
unum_compare(greater_ubound.right_bound,c[2].right_bound) ){
          // printf(" \n\nSplitting c3\n");

 s1_split[i] = s[0];
 s2_split[i] = s[1];
 s3_split[i] = s[2];
 s4_split[i] = s[3];
 s5_split[i] = s[4];
 s6_split[i] = s[5];

 c1_split[i] = c[0];
 c2_split[i] = c[1];
 c3_split[i] = ubound_ranges[i];
 c4_split[i] = c[3];
 c5_split[i] = c[4];
 c6_split[i] = c[5];

#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
```

```
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
                }
//If c4 is greater
else if(unum_compare(greater_ubound.left_bound,
c[3].left_bound)&& unum_compare(greater_ubound.right_bound,
c[3].right_bound) ){
//   printf(" \n\nSplitting c4\n");

s1_split[i] = s[0];
 s2_split[i] = s[1];
 s3_split[i] = s[2];
```

```
s4_split[i] = s[3];
s5_split[i] = s[4];
s6_split[i] = s[5];

c1_split[i] = c[0];
c2_split[i] = c[1];
c3_split[i] = c[2];
c4_split[i] = ubound_ranges[i];
c5_split[i] = c[4];
c6_split[i] = c[5];

#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
```

```
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
}
//If c5 is greater
else if(unum_compare(greater_ubound.left_bound,
c[4].left_bound)&& unum_compare(greater_ubound.right_bound,
c[4].right_bound) ){
// printf(" \n\nSplitting c5\n");
 s1_split[i] = s[0];
 s2_split[i] = s[1];
 s3_split[i] = s[2];
 s4_split[i] = s[3];
 s5_split[i] = s[4];
 s6_split[i] = s[5];

 c1_split[i] = c[0];
 c2_split[i] = c[1];
 c3_split[i] = c[2];
 c4_split[i] = c[3];
 c5_split[i] = ubound_ranges[i];
 c6_split[i] = c[5];

#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
```

```
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
#endif
                }
//If c6 is greater
else /* if(unum_compare(greater_ubound.left_bound,
c[6].left_bound)&& unum_compare(greater_ubound.right_bound,
c[6].right_bound) )*/{
// printf(" \n\nSplitting c6\n");

s1_split[i] = s[0];
 s2_split[i] = s[1];
 s3_split[i] = s[2];
 s4_split[i] = s[3];
 s5_split[i] = s[4];
 s6_split[i] = s[5];
```

```
 c1_split[i] = c[0];
 c2_split[i] = c[1];
 c3_split[i] = c[2];
 c4_split[i] = c[3];
 c5_split[i] = c[4];
 c6_split[i] = ubound_ranges[i];
#ifdef PRINT_SPLITS
printf("\n s1 split:");
ub2f_view(&s1_split[i],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("\t s2 split:");
ub2f_view(&s2_split[i],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
printf("\t s3 split:");
ub2f_view(&s3_split[i],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
printf("\t s4 split:");
ub2f_view(&s4_split[i],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
printf("\t s5 split:");
ub2f_view(&s5_split[i],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
printf("\t s6 split:");
ub2f_view(&s6_split[i],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
printf("\n c1 split:");
ub2f_view(&c1_split[i],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
printf("\t c2 split:");
ub2f_view(&c2_split[i],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
printf("\t c3 split:");
ub2f_view(&c3_split[i],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
printf("\t c4 split:");
ub2f_view(&c4_split[i],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
printf("\t c5 split:");
ub2f_view(&c5_split[i],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
printf("\t c6 split:");
ub2f_view(&c6_split[i],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
```

```
#endif
                }



//check if the the split satisfies the constraints:
 i.e If the resultant
ubounds encloses the right hand side solution, then enqueue it.
This split has passed but this might or
 might not be the required solution.
valid_comb = checkConstraintRobotarm(s1_split[i],s2_split[i],
s3_split[i],s4_split[i],s5_split[i],s6_split[i],c1_split[i],
c2_split[i],c3_split[i],c4_split[i],c5_split[i],c6_split[i],
&pass,&noInf);
if(valid_comb)
valid_cnt++;
 fn_eval++;
//Enqueue if valid_cnt is less than three. If valid_cnt is
equal to three, then don't enqueue these trials
if(valid_comb && valid_cnt<3) {

push(q,s1_split[i]);
push(q,s2_split[i]);
push(q,s3_split[i]);
push(q,s4_split[i]);
push(q,s5_split[i]);
push(q,s6_split[i]);
push(q,c1_split[i]);
push(q,c2_split[i]);
push(q,c3_split[i]);
push(q,c4_split[i]);
push(q,c5_split[i]);
push(q,c6_split[i]);
```

```
passed_cnt++;
//printf("\n");
                }

if(pass){
printf("\n pass=%d",pass);
 sols_s1[sols_cnt] = s1_split[i];
 sols_s2[sols_cnt] = s2_split[i];
 sols_s3[sols_cnt] = s3_split[i];
 sols_s4[sols_cnt] = s4_split[i];
 sols_s5[sols_cnt] = s5_split[i];
 sols_s6[sols_cnt] = s6_split[i];
 sols_c1[sols_cnt] = c1_split[i];
 sols_c2[sols_cnt] = c2_split[i];
 sols_c3[sols_cnt] = c3_split[i];
 sols_c4[sols_cnt] = c4_split[i];
 sols_c5[sols_cnt] = c5_split[i];
 sols_c6[sols_cnt] = c6_split[i];
sols_cnt+=1;
 }

if((exp > MAX_EXP)||(frac > MAX_FRAC))
break;


}
}
//reset number of ranges to 0 after generating 3 trial splits

        number_of_ranges = 0;
valid_cnt =0;
        tot_abs_width_s = 0;
        tot_abs_width_c = 0;
```

```
//    } //end of first for loop

//If nothing can be split, then queue these entries seperately
as a glayer interval containing float endpoints
if(s1_NO_SPLIT &&  s2_NO_SPLIT &&  s3_NO_SPLIT
&&  s4_NO_SPLIT && s5_NO_SPLIT &&  s6_NO_SPLIT
&& c1_NO_SPLIT &&  c2_NO_SPLIT &&c3_NO_SPLIT
&&  c4_NO_SPLIT &&  c5_NO_SPLIT &&  c6_NO_SPLIT){
//   printf("\n No ubound can be split further");
for(i=0;i<=5;i++){
fs[i].f_lb = lb_s[i];
fs[i].f_rb = rb_s[i];
fs[i].lb = sl_open[i];
fs[i].rb = sr_open[i];
pushfq(r,fs[i]);
}
for(i=0;i<=5;i++){
fc[i].f_lb = lb_c[i];
fc[i].f_rb = rb_c[i];
fc[i].lb = cl_open[i];
fc[i].rb = cr_open[i];
pushfq(r,fc[i]);
}
}

//If any one of the resultant ubounds is infinity,
and the environment is not warlpiri,then push all
the elements in the queue q to queue r and exit from loop
if((noInf==false)&& exp>0 && frac>0){
 while(!isEmpty(q)){
```

```
        printf("\n popping elements from q");
  //pop from queue q and push it to queue r
        for(i=0;i<=5;i++) {
           s[i] = pop(q);
           ub2f(&s[i],&lb_s[i],&rb_s[i],&sl_open[i],&sr_open[i]);
           }
        for(i=0;i<=5;i++) {
           c[i] = pop(q);
           ub2f(&c[i],&lb_c[i],&rb_c[i],&cl_open[i],&cr_open[i]);
           }
        for(i=0;i<=5;i++){
        fs[i].f_lb = lb_s[i];
        fs[i].f_rb = rb_s[i];
        fs[i].lb = sl_open[i];
        fs[i].rb = sr_open[i];
        pushfq(r,fs[i]);
        }

        for(i=0;i<=5;i++){
        fc[i].f_lb = lb_c[i];
        fc[i].f_rb = rb_c[i];
        fc[i].lb = cl_open[i];
        fc[i].rb = cr_open[i];
        pushfq(r,fc[i]);
        }
}
}
  if((exp > MAX_EXP)||(frac > MAX_FRAC))
        break;
   }
//when there is a valid combination that cannot be split
further or in other words reached the minimum ULP size,
```

```
then it must be in queue r
ubFloat_t fss1 ,fss2 ,fss3 ,fss4 ,fss5 ,fss6 ,fcc1 ,fcc2 ,
fcc3 ,fcc4 ,fcc5 ,fcc6 ;
exp_cnt=0, frac_cnt=0;
//If the number of solutions not found, then
check for the queue R
if((exp<MAX_EXP)||(frac<MAX_FRAC)){
 while(!isFqEmpty(r)){
for(i=0;i<=5;i++)
fss[i] = popfq(r);
for(i=0;i<=5;i++)
fcc[i] = popfq(r);

//If the exponent is increased once in the queue,
then it need not be increased again
//check if the ubounds need more exp/frac in the
current environment,if so increase the frac/exp
if((number_of_ranges == 0)){
if((fss[0].f_lb == _g_maxreal)|| (fss[0].f_rb == -_g_maxreal)
||(fss[1].f_lb == _g_maxreal) || (fss[1].f_rb == -_g_maxreal)
||(fss[0].f_lb == -_g_smallsubnormal)||
(fss[0].f_rb == _g_smallsubnormal)||
(fss[1].f_lb == -_g_smallsubnormal)
|| (fss[1].f_rb == _g_smallsubnormal)
|| (fss[0].f_lb== -INFINITY) || (fss[1].f_lb == -INFINITY)
|| (fss[0].f_rb == INFINITY) || (fss[1].f_rb == INFINITY)
||(fss[2].f_lb == _g_maxreal)|| (fss[2].f_rb == -_g_maxreal) ||
(fss[3].f_lb == _g_maxreal) || (fss[3].f_rb == -_g_maxreal) ||
(fss[2].f_lb == -_g_smallsubnormal)||
(fss[2].f_rb == _g_smallsubnormal) ||
(fss[3].f_lb == -_g_smallsubnormal) ||
(fss[3].f_rb == _g_smallsubnormal) ||
```

```
(fss[2].f_lb== -INFINITY) || (fss[3].f_lb == -INFINITY) ||
(fss[2].f_rb == INFINITY) ||     (fss[3].f_rb == INFINITY) ||
(fss[4].f_lb == _g_maxreal)|| (fss[4].f_rb == -_g_maxreal) ||
(fss[5].f_lb == _g_maxreal) || (fss[5].f_rb == -_g_maxreal) ||
(fss[4].f_lb == -_g_smallsubnormal)||
(fss[4].f_rb == _g_smallsubnormal) ||
(fss[5].f_lb == -_g_smallsubnormal) ||
(fss[5].f_rb == _g_smallsubnormal) ||
(fss[4].f_lb== -INFINITY) || (fss[5].f_lb == -INFINITY) ||
fss[4].f_rb == INFINITY) ||      (fss[5].f_rb == INFINITY) ||
(fcc[0].f_lb == _g_maxreal)|| (fcc[0].f_rb == -_g_maxreal) ||
(fcc[1].f_lb == _g_maxreal) || (fcc[1].f_rb == -_g_maxreal) ||
(fcc[0].f_lb == -_g_smallsubnormal)||
(fcc[0].f_rb == _g_smallsubnormal)||
(fcc[1].f_lb == -_g_smallsubnormal)
|| (fcc[1].f_rb == _g_smallsubnormal) ||
(fcc[0].f_lb== -INFINITY) || (fcc[1].f_lb == -INFINITY) ||
(fcc[0].f_rb == INFINITY) ||     (fcc[1].f_rb == INFINITY) ||
(fcc[2].f_lb == _g_maxreal)|| (fcc[2].f_rb == -_g_maxreal) ||
(fcc[3].f_lb == _g_maxreal) || (fcc[3].f_rb == -_g_maxreal) ||
(fcc[2].f_lb == -_g_smallsubnormal)||
(fcc[2].f_rb == _g_smallsubnormal) ||
(fcc[3].f_lb == -_g_smallsubnormal) ||
(fcc[3].f_rb == _g_smallsubnormal) ||
(fcc[2].f_lb== -INFINITY) || (fcc[3].f_lb == -INFINITY)
 || (fcc[2].f_rb == INFINITY)
|| (fcc[3].f_rb == INFINITY) ||   (fcc[4].f_lb == _g_maxreal)
|| (fcc[4].f_rb == -_g_maxreal)
|| (fcc[5].f_lb == _g_maxreal) || (fcc[5].f_rb == -_g_maxreal)
 ||
(fcc[4].f_lb == -_g_smallsubnormal)     ||
(fcc[4].f_rb == _g_smallsubnormal)
```

```
||   (fcc[5].f_lb == -_g_smallsubnormal) ||
(fcc[5].f_rb == _g_smallsubnormal) ||
(fcc[4].f_lb== -INFINITY) || (fcc[5].f_lb == -INFINITY)
|| (fcc[4].f_rb == INFINITY)
|| (fcc[5].f_rb == INFINITY) || noInf==false)
{

printf("\n noInf is false");
if((exp_cnt==0) && (exp < MAX_EXP)){
 printf("\n---------------- Statistics for environment
 (%d,%d)----------------",exp,frac);
 printf("\n size of queue r
 (contains all minimum ULP splits) is:");
 displayfqQsize(r);
 printf("\n Total function evaluations=%d",fn_eval);
 printf("\n Total passed evaluations=%d",passed_cnt);
 printf("\n----------------End of statistics for environment
 (%d,%d)------------",exp,frac);
 printf("\n Printing the content of the queue for environment
 (%d,%d)",exp,frac);
 printf("\n Queue q is:");
 display(q);
 printf("\n Queue r is:");
 displayfq(r);
 printf("\n Increasing the exponent");
 exp+=1;
 printf("\n setting environment to (%d,%d)",exp,frac);
 exp_cnt++;
}
//   }

//If the fraction count is zero and solution is not found,
```

```
then increase the fraction
else if((frac_cnt == 0)&&(sols_cnt==0)&&(frac < MAX_FRAC)){
 printf("\n—————— Statistics for environment
 (%d,%d)————————————————",exp,frac);
 printf("\n size of queue r (contains all
 minimum ULP splits) is:");
 displayfqQsize(r);
 printf("\n Total function evaluations=%d",fn_eval);
 printf("\n Total passed evaluations=%d",passed_cnt);
 printf("\n——————End of statistics for environment
 (%d,%d)————————————",exp,frac);
 printf("\n Printing the content of the queue for environment
 (%d,%d)",exp,frac);
 printf("\n Queue q is:");
 display(q);
 printf("\n Queue r is:");
 displayfq(r);
printf("\n Increasing the fraction");
frac+=1;
printf("\n setting environment to (%d,%d)",exp,frac);
frac_cnt++;
}
set_env(exp,frac);
for(i=0;i<=5;i++){
x2u(fss[i].f_lb,&gb_fss[i].left_bound);
x2u(fss[i].f_rb,&gb_fss[i].right_bound);
gb_fss[i].left_open = fss[i].lb;
gb_fss[i].right_open = fss[i].rb;
//Get the ubounds from gbounds
get_ubound_result_from_gbound(&ub_fss[i], &gb_fss[i]);
push(q,ub_fss[i]);
}
```

```
for(i=0;i<=5;i++){
x2u(fcc[i].f_lb,&gb_fcc[i].left_bound);
x2u(fcc[i].f_rb,&gb_fcc[i].right_bound);
gb_fcc[i].left_open = fcc[i].lb;
gb_fcc[i].right_open = fcc[i].rb;
//Get the ubounds from gbounds
get_ubound_result_from_gbound(&ub_fcc[i], &gb_fcc[i]);
push(q,ub_fcc[i]);
}}}
if((exp<MAX_EXP)||(frac<MAX_FRAC))
goto REPEATPRUNING;
printf("\n Queue r is");
displayfq(r);
printf("\n The solutions are:");
for(k=0;k<sols_cnt;k++){
printf("\n s1 \t");
ub2f_view(&sols_s1[k],&lb_s1,&rb_s1,&s1l_open,&s1r_open);
printf("s2");
ub2f_view(&sols_s2[k],&lb_s2,&rb_s2,&s2l_open,&s2r_open);
  printf("\n s3 \t");
  ub2f_view(&sols_s3[k],&lb_s3,&rb_s3,&s3l_open,&s3r_open);
  printf("s4");
  ub2f_view(&sols_s4[k],&lb_s4,&rb_s4,&s4l_open,&s4r_open);
  printf("\n s5 \t");
  ub2f_view(&sols_s5[k],&lb_s5,&rb_s5,&s5l_open,&s5r_open);
  printf("s6");
  ub2f_view(&sols_s6[k],&lb_s6,&rb_s6,&s6l_open,&s6r_open);
  printf("c1");
  ub2f_view(&sols_c1[k],&lb_c1,&rb_c1,&c1l_open,&c1r_open);
  printf("c2");
  ub2f_view(&sols_c2[k],&lb_c2,&rb_c2,&c2l_open,&c2r_open);
  printf("\n c3 \t");
```

```
  ub2f_view(&sols_c3[k],&lb_c3,&rb_c3,&c3l_open,&c3r_open);
  printf("c4");
  ub2f_view(&sols_c4[k],&lb_c4,&rb_c4,&c4l_open,&c4r_open);
  printf("\n c5 \t");
  ub2f_view(&sols_c5[k],&lb_c5,&rb_c5,&c5l_open,&c5r_open);
  printf("c6");
  ub2f_view(&sols_c6[k],&lb_c6,&rb_c6,&c6l_open,&c6r_open);
 }
printf("\n Total function evaluations=%d",fn_eval);
printf("\n Total passed evaluations=%d",passed_cnt);
return 0; }
```

SPLIT ROUTINE IMPLEMENTED FOR UNUMS

```
//split the ubound into the specified number
of splits
int splitub(double lb, double rb,bool left_open,bool right_open,
ubound_t ub,gbound_t ranges[],ubound_t ranges_ub[] )
{
//If both the endpoints of a gbound are the same and
they are closed intervals, then nothing to split
 gbound_t temp;

// printf("\n entering inside split routine");
if((lb==rb) && !left_open && !right_open){
//  printf("\n both closed");
  return 0;}

//If both are closed and are different unum pairs, then copy
unum1 to 1st split and unum2 to 2nd split
else if((lb!=rb) && !left_open && !right_open)
  {
```

```
//    printf("\n both are closed and diff");
    ranges[0].left_bound = ub.left_bound;
    ranges[0].right_bound = ub.left_bound;
    ranges[0].left_open = 0;
    ranges[0].right_open = 0;

    ranges[1].left_bound = ub.left_bound;
    ranges[1].right_bound = ub.right_bound;
    ranges[1].left_open = 1;
    ranges[1].right_open = 1;

    ranges[2].left_bound = ub.right_bound;
    ranges[2].right_bound = ub.right_bound;
    ranges[2].left_open = 0;
    ranges[2].right_open = 0;

    return 3;
    }


  //If the left endpoint is -Infinity and the right endpoint
  is neg maxreal, nothing to split
   else if((isinf(lb)==-1)&& (rb==-_g_maxreal)&&
   left_open && right_open){
  //  printf("\n Can't split anymore, 'negative many' region");
        return 0;
   }
   else if((isinf(lb)==-1)&& (rb==-0)&& left_open
  && right_open){

    ranges[0].left_bound = _g_minrealu;
    ranges[0].right_bound = _g_minrealu;
```

```
  ranges[0].left_open = 1;
  ranges[0].right_open = 1;

  x2u(-_g_maxreal, &ranges[1].left_bound);
  x2u(-_g_maxreal, &ranges[1].right_bound);
  ranges[1].left_open = 0;
  ranges[1].right_open = 0;

  ranges[2].left_bound = ranges[1].right_bound;
  get_gbound_from_unum(&temp,&ub.right_bound);
  ranges[2].right_bound = temp.right_bound;
  ranges[2].left_open = 1;
  ranges[2].right_open = 1;

  return 3;
 }
//If the left endpoint is -Infinity and the right
endpoint is not equal to -maxreal, then split accordingly
 else if((isinf(lb)==-1)&& (rb!=-_g_maxreal)
 && left_open && right_open){
//   printf("\n -Infinity to something not maxreal");
//For this case, if its a warlpiri environment,
then split at 0 instead of splitting at minrealu
  if(_g_utagsize == 1){
//   printf("utagsize is 1");
  ranges[0].left_bound = _g_neginfu;
  ranges[0].right_bound = _g_zero;
  ranges[0].left_open = 1;
  ranges[0].right_open = 1;

  x2u(0, &ranges[1].left_bound);
  x2u(0, &ranges[1].right_bound);
```

```
  ranges[2].left_bound = ranges[1].right_bound;
  get_gbound_from_unum(&temp,&ub.right_bound);
  ranges[2].right_bound = temp.right_bound;
  ranges[2].left_open = 1;
  ranges[2].right_open = 1;

  return 3;

  }
  else{
//   printf("\n Entering in the correct condn");
  ranges[0].left_bound = _g_minrealu;
  ranges[0].right_bound = _g_minrealu;
  ranges[0].left_open = 1;
  ranges[0].right_open = 1;

  x2u(-_g_maxreal, &ranges[1].left_bound);
  x2u(-_g_maxreal, &ranges[1].right_bound);
  ranges[1].left_open = 0;
  ranges[1].right_open = 0;

  ranges[2].left_bound = ranges[1].right_bound;
  get_gbound_from_unum(&temp,&ub.right_bound);
  ranges[2].right_bound = temp.right_bound;
  ranges[2].left_open = 1;
  ranges[2].right_open = 1;

  return 3;
       }
  }
//cannot split if its 'positive' many region
```

```
else if((isinf(rb)==1) && (lb==_g_maxreal)&&
left_open && right_open){
//   printf("\n can't split anymore, 'positive many region'");
       return 0;
}

else if((isinf(rb)==1) && (lb!=_g_maxreal)&&
left_open && right_open){
 //printf("\n the final condn");
 ranges[0].left_bound = ub.left_bound;
 x2u(_g_maxreal, &ranges[0].right_bound);
 ranges[0].left_open = 1;
 ranges[0].right_open = 1;

 x2u(_g_maxreal, &ranges[1].left_bound);
 x2u(_g_maxreal, &ranges[1].right_bound);
 ranges[1].left_open = 0;
 ranges[1].right_open = 0;

 ranges[2].left_bound = _g_maxrealu;
 ranges[2].right_bound = _g_maxrealu;
 ranges[2].left_open = 1;
 ranges[2].right_open = 1;
 return 3;
        }

else if(!left_open && right_open){
//   printf("one open other closed");
 ranges[0].left_bound = ub.left_bound;
 ranges[0].right_bound = ub.left_bound;
 ranges[0].left_open = 0;
 ranges[0].right_open = 0;
```

```
ranges[1].left_bound = ub.left_bound;
ranges[1].right_bound = ub.right_bound;
ranges[1].left_open = 1;
ranges[1].right_open = 1;
return 2;
}

else if(left_open && !right_open){
// printf("\n one open other closed");
ranges[0].left_bound = ub.left_bound;
ranges[0].right_bound = ub.right_bound;
ranges[0].left_open = 1;
ranges[0].right_open = 1;


ranges[1].left_bound = ub.right_bound;
ranges[1].right_bound = ub.right_bound;
ranges[1].left_open = 0;
ranges[1].right_open = 0;
return 2;
}



else{
//  printf("\n final condn");
//compute the exact coverage for the endpoints
and convert it into a ubound/unum
x2u((lb+rb)/2, &ranges_ub[0].right_bound);
//obtain the gbound result from the ubound
get_gbound_from_unum(&temp,&ranges_ub[0].right_bound);
```

```
//compute the actual double values representing
the gbound endpoints

double gm1 = u2f(temp.left_bound);
double gm2 = u2f(temp.right_bound);
//printf("\n gm2=%f",gm2);

if((gm1 > lb)&& (gm2!=rb))
{
ranges[0].left_open = 1;
ranges[0].right_open = 1;
x2u(lb,&ranges[0].left_bound);
ranges[0].right_bound = temp.left_bound;

ranges[1].left_bound = temp.left_bound;
ranges[1].right_bound = temp.left_bound;
ranges[1].left_open = 0;
ranges[1].right_open = 0;

ranges[2].left_bound = temp.left_bound;
x2u(rb, &ranges[2].right_bound);
get_gbound_from_unum(&temp,&ranges[2].right_bound);
ranges[2].right_bound = temp.right_bound;
ranges[2].left_open = 1;
ranges[2].right_open = 1;
return 3;
}
else if((gm2 < rb)&& (gm1!=lb))
{
printf("\n entering correct optn");
ranges[0].left_open = 1;
ranges[0].right_open = 1;
```

```
x2u(lb,&ranges[0].left_bound);
ranges[0].right_bound = temp.right_bound;
ranges[1].left_bound = temp.right_bound;
ranges[1].right_bound = temp.right_bound;
ranges[1].left_open = 0;
ranges[1].right_open = 0;
ranges[2].left_bound = ranges_ub[0].right_bound;
x2u(rb, &ranges[2].right_bound);
ranges[2].left_open = 1;
ranges[2].right_open = 1;
return 3;
}
else{
//Enters this case is when the endpoints lb and rb match gm1
and gm2. No more splitting possible for this case
//printf("\n No more splitting possible for this case");
return 0;
}
}
}
```