Fakultät für Maschinenwesen

Lehrstuhl für Flugsystemdynamik

Development of Elementary Mathematics Functions in an Avionics Context

Dipl.-Ing. Kajetan Nürnberger

Vollständiger Abdruck der von der Fakultät für Maschinenwesen

der Technischen Universität München

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.) genehmigten Dissertation.

Vorsitzender: Prof. Dr.-Ing. Manfred Hajek

Prüfer der Dissertation: 1. Prof. Dr.-Ing. Florian Holzapfel

2. Prof. Dr. Marco Caccamo

Die Dissertation wurde am 12.04.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Maschinenwesen am 29.09.2019 angenommen.

Acknowledgement

This thesis is the final output after working for five years at the Institute of Flight System Dynamics at TU Munich plus some long evenings after leaving the university. The time at the institute was one of the most inspiring times of my life. I did not only gain many new insights in the area of my research, but also in many related fields as handling of projects or communicating in complex organizations. Although working at the institute was a really interesting and often quite funny time, there were also parts that were challenging. Without the support of various people, writing this thesis would not have been possible. Therefore, I would like to express my gratitude to all of these here.

First of all, I would like to thank Professor Florian Holzapfel. He gave me the chance to work at the institute on really interesting technology. During the time at the institute, I had the chance to gain insights into different research fields. I appreciate a lot that Florian gave me the possibility to freely decide on the research area this thesis is dealing with.

A big thank you to all my colleagues at the institute who contributed to the flight control model which served as a real-world example for the research work presented in this thesis. Special thanks to our coffee group at various locations. During the coffee breaks, several interesting discussions took place. In case I was suck at a problem, this brake gave the necessary distraction. Really important for me is that during the time at the institute, I met a lot of new colleagues who became good friends in the meantime.

I would like to express my gratitude to Professor Marco Caccamo for taking the task of the second examiner shortly after he joined TU Munich. Thanks also to Professor Manfred Hajek who acted as the chairman of the committee after the original chairman has left the TU Munich.

Thanks to everyone who gave comments on the work which showed me the right spots where further improvement was necessary.

Finally, I want to thank my family who always supported me during the time the thesis was constructed. Without their support, it would not have been possible to finalize the thesis.

Abstract

Model based development approaches enable an early test of the developed algorithm in a simulation environment. To get the system to the actual target platform, auto code generation can be used. As the auto generated code relies on external components, the target behavior can deviate from the behavior in the simulation, although the automatically generated code does not contain any error. This problem is addressed in this thesis for a flight control system where the external components are elementary math functions.

The main challenges addressed during the development of the math functions are an efficient execution and a formal verification of the algorithms. The efficient implementation can reduce the overall latency of the control loop and, therefore, lead to a better overall performance. The formal verification mainly addresses the precision of the implementation. Many available library implementations lack of such a precision proof. In case it is available the goal is to show the accuracy up to the last bit, which is not desired for an embedded system. To reach the goal of an efficient formally verified implementation, the input ranges of the elementary math functions are adapted. This is motivated based on functional considerations and verified by a runtime error analysis of the integrated software. In order to successfully execute this analysis with no false alarms, the elementary math functions are replaced by specific stubs in the analysis. These stubs abstract the behavior of the original function in a way that sound result with a low number of false alarms is achieved in the runtime error analysis. The validity of all properties of the stubs are shown via formal proofs along with the verification of the reached precision. In order to reach a high precision in an efficient manner advanced polynomial approximations are used and executed using double precision floating-point arithmetic. To ensure the portability between the target and the development host architecture, hardware specific functions are omitted. In case it is not possible to omit functions custom to the hardware, these are captured in a separate abstraction layer to ensure an easy portability. Approximations are given for the trigonometric functions, the square root and the power function with integer exponents. The proposed algorithms can completely be implemented on any processor supporting double precision floating-point arithmetic, as they do not rely on special hardware features. Different approximation methods are evaluated and the actually selected implementation is based on the worst case execution time.

To determine the worst case timing a static analysis method is used. As the analysis during the early development phase is limited to a small function of the overall software, it is assumed that the function is available in the cache memory for the analysis. The validity of this assumption is later shown in an execution time analysis of the complete software. Here it is shown that locking the elementary math functions to the cache statically brings a benefit of the analyzed worst case time and the measured maximum and average execution time. With the use of the WCET analysis results further parts of the software can be identified as candidates for a cache locking. By statically locking these parts, the WCET, the measured maximum and average execution time can be further reduced.

Table of Contents

List of Figures ... xi
List of Tables ... xiii
List of Acronyms ... xv
List of Symbols ... xvii
1 Introduction ... 1
1.1 Motivation and Background ... 1
1.2 State of the Art ... 5
1.3 Objectives ... 9
1.4 Contributions ... 10
1.5 Outline ... 11
2 Fundamentals ... 13
2.1 Nomenclature ... 13
2.2 Floating-Point Arithmetic ... 14
2.3 Worst Case Execution Time ... 17
3 Demonstration Environment Description ... 21
3.1 Flight Control Computer Hardware Overview ... 21
3.2 Application Software Overview ... 22
3.3 Model-Based Development Approach ... 23
3.4 Base Software ... 25
3.5 Software Statistics ... 26
4 Library Development Process ... 29
4.1 Workflow ... 29
4.2 Architecture and Implementation Considerations ... 35
4.3 Relation to Certification Aspects ... 38
5 Trigonometric Functions ... 41
5.1 Sine and Cosine ... 41
5.1.1 Input Restrictions ... 41
5.1.2 Architecture ... 41
5.1.3 Range Reduction ... 43
5.1.4 Core Algorithm ... 47
5.1.5 Algorithm Selection and Implementation Considerations ... 51
5.1.6 Function Stub for the Runtime Error Analysis ... 54
5.1.7 Precision Differential Quotient ... 56
5.2 Tangent ... 59

5.2.1 Input Restrictions 59
5.2.2 Architecture 59
5.2.3 Range Reduction 61
5.2.4 Core Algorithm 62
5.2.5 Argument Reconstruction 63
5.2.6 Algorithm Selection 65
5.2.7 Function Stub for the Runtime Error Analysis 66
5.3 Arctangent 66
5.3.1 Input Restrictions 67
5.3.2 Architecture 67
5.3.3 Range Reduction 68
5.3.4 Core Algorithm 69
5.3.5 Algorithm Selection and Implementation Considerations 70
5.3.6 Function Stub for the Runtime Error Analysis 75
5.4 Arcsine and Arccosine 76
5.4.1 Input Restrictions 76
5.4.2 Architecture 76
5.4.3 Range Reduction 77
5.4.4 Core Algorithm 77
5.4.5 Arcus Cosine 81
5.4.6 Algorithm Selection and Implementation Considerations 84
5.4.7 Function Stub for the Runtime Error Analysis 85
6 Square Root 87
6.1 Function Implementation 87
6.2 Function Stub for the Runtime Error Analysis 94
7 Exponentiation 95
7.1 Function Implementation 95
7.2 Function Stub for the Runtime Error Analysis 98
8 Runtime Error Analysis 99
8.1 Adaptation of Model 100
8.2 Improved by Annotations 104
9 Execution Time Analysis 111
9.1 First Variant 114
9.2 Second Variant 115
9.3 Results 117
10 Conclusion and Future Work 121

10.1 Conclusion 121
10.2 Future Work 122
Bibliography I
Appendix VII
A Polynomial Coefficients VII

List of Figures

Figure 1-1: Simulink Single vs. Double Sine 2
Figure 1-2: Result of the Model Shown in Figure 1-1 2
Figure 1-3: The Sine Function in the Given Floating-Point Number Format 4
Figure 2-1: 22.5 Expressed as Floating-Point Number with Decimal and Binary Base 14
Figure 2-2: Measured vs. Possible Execution Times vs. WCET Analysis Result 18
Figure 2-3: Cache Structure 19
Figure 3-1: Research Aircraft from TUM FSD 21
Figure 3-2: Flight Control Computer Structure [57] 21
Figure 3-3: Flight Control Modules Overview [57] 23
Figure 3-4: FCC SW Sequence Diagram [57] 25
Figure 4-1: Workflow Overview 29
Figure 4-2: Relative Error: `fpminimax` Approximation vs. Rounded Remez Approximation 32
Figure 4-3: Relative Error Taylor Approximation 32
Figure 4-4: Exemplary PIL Result 34
Figure 4-5: Library Architecture Overview 35
Figure 5-1: Sine and Cosine in the Unit Circle 42
Figure 5-2: Cosine close to $\pi/2$ 45
Figure 5-3: Flow Diagram of the Cosine Sollya Script 48
Figure 5-4: Cosine Approximation Error and Total Error 49
Figure 5-5: Visualization of the Different Approximations 51
Figure 5-6: Relative Error of the Different Sine / Cosine Approximations 52
Figure 5-7: WCET and Precision of the Cosine Variants 53
Figure 5-8: Motivation for an Enhanced Cosine Stub 55
Figure 5-9: Sine and Cosine Absolute Errors 57
Figure 5-10: Precision of Different Tangent Approximation Polynomials 60
Figure 5-11: Analyzed WCET for the Tangent Function 66
Figure 5-12: Visualization of the Arcus Tangent Approximation Intervals 68
Figure 5-13: Different Arcus Tangent Approximation Polynomials 70
Figure 5-14: Arcus Tangent Range Reduction for s = 2 Variant 71
Figure 5-15: Optimal Binary search for the s = 3 Variant 72
Figure 5-16: None Optimal Binary Search for the s = 4 Variant 72
Figure 5-17: Analyzed WCET for Arcus Tangent Function 73
Figure 5-18: Flow Graph of Sollya Script to Determine Ranges and Number of Approximations for the Arcsine 78

Figure 5-19: Sollya Script for the Calculation of the Arcus Cosine Relative Approximation Error 83
Figure 5-20: Analyzed WCET for Arcus Sine Function 84
Figure 7-1: Simulink Implementation of a Polynomial in Horner Scheme 97
Figure 8-1: Implementation of the Default “Saturation Dynamic” 100
Figure 8-2: Implementation of the Custom “Saturation Dynamic” 101
Figure 9-1: Analyzed WCET over Available Cache Ways 113
Figure 9-2: Execution Time Distribution 118

List of Tables

Table 2-1: Bit Representation of a Single Number 15
Table 2-2: Bit Representation of a Double Number 15
Table 3-1: Usage of Library Functions 27
Table 5-1: Mapping from the Range Reduction to the Cosine Approximation for $[0, \pi/2]$ 42
Table 5-2: Mapping from the Range Reduction to the Cosine Approximation for $[0, \pi/4]$ 43
Table 5-3: Precisions Reached by the Polynomial Approximations of the Tangent 63
Table 5-4: Precisions Reached for the Complete Input Range for `tan_pi32` Variants 65
Table 5-5: Precisions Reached by `atan_pi12_17` on Different Intervals 74
Table 5-6: Conditions for the `atan2` Calculation 74
Table 5-7: Range of Linear Approximation of the Arcus Sine 78
Table 5-8: Polynomial Grades for Arcus Sine Approximation. Range: $[2 - 26, 0.854]$ 79
Table 5-9: Precisions Reached by Direct Arcus Sine Implementation 80
Table 5-10: Precisions Reached by the Arcus Sine Implementation via the Arcus Tangent..81
Table 6-1: Mapping of 0x5fe80000 to the Sig, Exponent and Significant 89
Table 7-1: Precision of the Power Function 96
Table 8-1: Signals Annotated after the mn_LatSig Function 107
Table 9-1: Locked Library Functions 115
Table 9-2: Analyzed WCET and Measured Time 117
Table 9-3: Improvements Achieved via Cache Locking 118

List of Acronyms

Acronym	Description
ATOL	Automatic Takeoff and Landing
C	C Programming Language
CAN	Controller Area Network
CPU	Central Processing Unit
CR	Correctly Rounded
DAL	Design Assurance Level
DDR	Double Data Rate
FIFO	First In First Out
FMA	Fused Multiply Add
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FSD	Institute of Flight System Dynamics
HIL	Hardware In the Loop
I/O	Input / Output
ICD	Interface Control Document
ILP	Integer Linear Programming
INF	Infinity
KB	Kilo Byte
LOC	Lines of Code
LRU	Least Recently Used
LSB	Least Significant Bit
N/A	Not Applicable
NaN	Not a Number
OSI	Open System Interconnection Model
PIT	Periodic Interrupt Timer
PLRU	Pseudo Least Recently Used
PowerPC	Performance optimization with enhanced RISC Performance Chip
RAM	Random Access Memory
SDRAM	Synchronous Dynamic RAM
SIMD	Single Instruction Multiple Data
SLCI	Simulink Code Inspector
SLOC	Source Lines of Code
TQL	Tool Qualification Level
TUM	Technical University Munich
ulp	unit in the last place
V&V	Validation and Verification
WCET	Worst Case Execution Time

List of Symbols

Operators	
Symbol	**Description**
$\oplus$	Floating-point summation
$\ominus$	Floating-point different
$\otimes$	Floating-point product
$\oslash$	Floating-point division
$\lfloor x \rceil$	Round to nearest integer of x
$\lfloor x \rfloor$	Round to largest integer smaller or equal than x (Floor Function)
$\lceil x \rceil$	Round to smallest integer greater or equal than x (Ceil Function)
$\tilde{x}$	Floating-point representative of x
Indices	
Symbol	**Description**
apr	Approximation
rnd	Rounding
red	Range Reduction
tot	Total
rel	Relative
abs	Absolute

1 Introduction

1.1 Motivation and Background

Modern flight control algorithms rely on various elementary mathematical functions. As compilers normally provide a library implementing these functions, many developers do not really care about how the desired functionality is achieved, and just use one of the available library implementations. In the context of a safety critical system, this becomes more complex. The certification authorities give guidance in [1] how to deal with such libraries. Here it is clearly stated that libraries are treated as any other part of airborne software and that all DO-178 [2] objectives also apply to libraries used in systems with a high assurance level. Therefore, it is not sufficient that just the object code of the libraries is used. Software requirements, a software-architecture, and a design with all the corresponding validation and verification (V&V) activities are necessary. The necessity that all DO-178 objectives must be fulfilled is quite obvious. Although, the compiler supplied libraries are used in various systems, this is not sufficient to proof the correct functionality in the system at hand. As it cannot be ensured that every applicant of the library reports problems with the libraries, the frequent use might not be a benefit at all. Many libraries contain several branches. Therefore, a compliant testing of the libraries is also not possible without a proper design of each function. This also must be considered in further V&V steps like a runtime error analysis or the worst case execution time (WCET) analysis. Depending on the implementation of a function there might be corner cases, which take significantly more time to calculate than the average. For a proper determination of the WCET also these cases must be considered.

Especially for the libraries implementing basic mathematical functions like the sine, there might also be a lack of specification. Many systems dealing with real world numbers today use floating-point arithmetic. The calculation is normally based on the standard IEEE 754 [3]. That standard defines different floating-point formats, rounding modes, and the behavior of basic operations, but, with one exception, the standard does not define the behavior of basic floating-point functions. This exception is the square root. The square root is the only basic function, which is required to be correctly rounded in order to comply with the standard. For the square root it is much easier to implement and proof a correctly rounded algorithm. Therefore, this might be the reason that this function is included in the standard. There is a little change in the standard between the editions from 1985 [4] and 2008 [3]. For most basic math functions the calculations of a correctly rounded result is quite computation intensive for certain cases. With the available computational power in 1985 it was unrealistic to implement such a function. For that reason, the only elementary function mentioned in the 1985 edition is the square root. As the computational power increased a lot in the past decades, the 2008 edition now also contains a recommendation for several other functions, which should be implemented correctly rounded. However, as it is still only a recommendation, correctly rounded functions are still not common. This leads to the fact that the calculation of a sine might lead to significantly different solutions in certain corner cases when different systems or compilers are used. In [5] an overview is given for the results of the sine on many different systems. Here the results for the same input value vary in a range $[-0.852, 0.874]$. This is almost the complete return range of the sine function. In [6] an approach to generate automated test stimuli for support functions is presented. In this context errors up to a value of $3.8e-06$ are found for a sine function. This error is significantly larger than the precision of the used double precision floating-point data types.

Most parts of the software used as demonstration environment in this thesis is auto generated out of a Simulink model. For this reason, the behavior of the elementary math functions in the Simulink environment are of special interest. The sine function used in current editions of Simulink is not correctly rounded. For the single variant of the function this can be shown with a simple example model which is displayed in Figure 1-1.

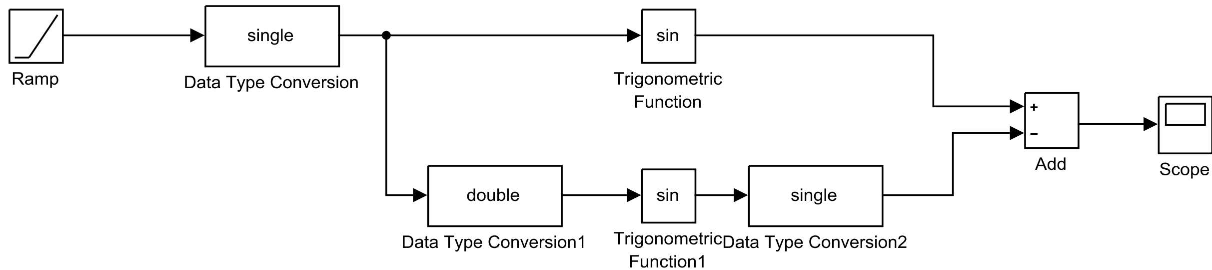

Figure 1-1: Simulink Single vs. Double Sine

As input signal to the sine a simple ramp is used. Afterwards a type conversion follows in order to ensure that the signal is single. On the lower signal path a conversion back to double follows. As every single number is exactly representable also in double precision, the conversion actually has no effect to the value of the signal. However, the conversion implies that the sine in the lower signal path is the double variant. As in the upper path the input is a single number, the single variant is used. In the lower path the result of the sine is then converted to single again. In case of a correctly rounded sine function both values should be exactly the same. The simulation was executed using MATLAB 2016b 64bit on Windows 7 running on a computer using an Intel Core i5-3570. As the graph in Figure 1-2 shows in this case, the implementation is not correctly rounded.

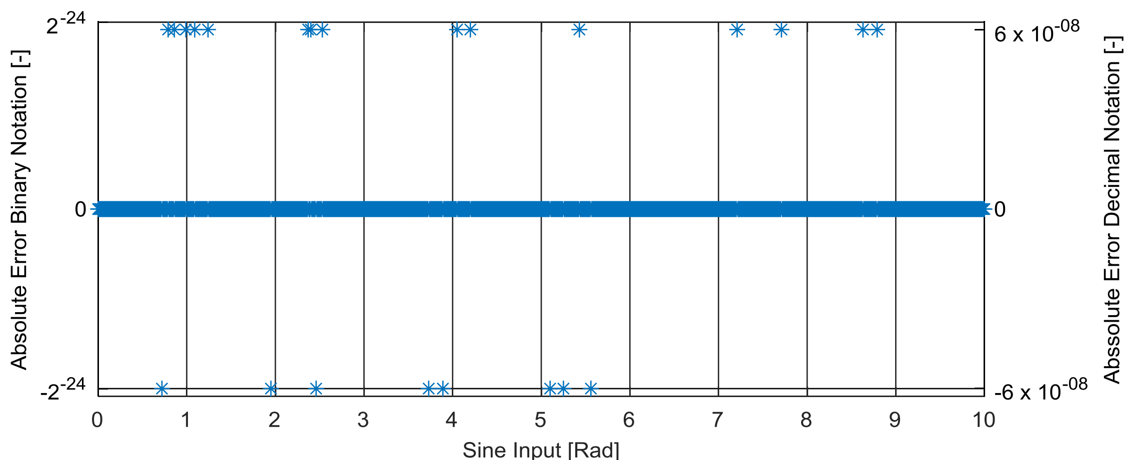

Figure 1-2: Result of the Model Shown in Figure 1-1

Due to the lack of standardization a reasonable precision that shall be reached by the basic math function needs to be defined. The computer science often tries to achieve correctly rounded functions. But, as embedded systems normally serve a dedicated purpose in an electrical or electromechanical system the question arises if a correctly rounded function is a goal worth looking at. One problem here might be a different understanding of floating-point numbers between computer science and a normal engineer. This problem is shown in the following paragraphs. Some examples from an industry perspective can also be found in [7].

Some element functions define a mapping from big numbers to small numbers. Therefore, the unit in the last place (ulp) for the floating-point function value $\widetilde{f(\tilde{x})}$ might be smaller than for the floating-point input $\tilde{x}$. So, in the following expression the absolute value of n can be greater than one.

$$f\left(\tilde{x} + ulp(\tilde{x})\right) = \widetilde{f(\tilde{x})} + n * ulp(\widetilde{f(\tilde{x})}) \tag{1-1}$$

In order to visualize this, a floating-point format with a decimal base and a minimum exponent of minus one is introduced. The number of digits in the mantissa is limited to three. The same format is also used in the following paragraphs. The representative of π in this number system is 3.14. As $ulp(3.14)$ equals to 0.01, the successor of 3.14 is 3.15. As discussed in the following and shown in Figure 1-3, the sine of 3.14 resolves to $0.02 * 10^{-1}$. The sine of 3.15 equals to $-0.08 * 10^{-1}$. For both results the ulp is $0.01 * 10^{-1}$. With this example equation (1-1) resolves to:

$$sin\left(3.14 + ulp(3.14)\right) = \sin(3.14) - 10 * ulp(sin(3.14)) \tag{1-2}$$

So in this case the absolute value of n is ten. As $\tilde{x}$ is a floating-point number, it is normally subject to rounding. In the example above all real numbers in the range of $[3.135.3145]$ would be rounded to 3.14. For the floating-point format at hand, eleven different floating-point values for the sine exist for real number inputs in the range of $[3.135{,}3.145]$. So in order to define a unique definition of $\widetilde{f(\tilde{x})}$ in general the computer science defines the floating-point representative $\tilde{x}$ of x as exact. Therefore, it defines $\widetilde{f(\tilde{x})} = \widetilde{f(x)}$, where $\tilde{x} - x = 0$ but the precision of x is so high that exactly one $\widetilde{f(x)}$ with the same precision as $\tilde{x}$ can be allocated to the input.

From an engineering point of view this might lead to a misunderstanding, which is presented here. First of all the function argument might be the result of previous calculations and might be subject to rounding. This leads to the fact that the above assumption that $\tilde{x} - x = 0$ is not valid, as $\tilde{x}$ was rounded before. Therefore, also if the elementary function is correctly rounded, the overall result might not be correctly rounded. Also, characteristic values given as a floating-point constant might be subject to rounding. So, a correctly rounded function might not return the expected result for characteristic values as the machine precision of the special value will not be high enough to return exactly the expected result. An example is the sine of the double representative of π. The user might expect this value would be zero, but a value in the range of 10^{-16} is returned. To demonstrative visualize this effect, the floating-point format as introduced before is used. In Figure 1-3 the sine close to π is visualized.

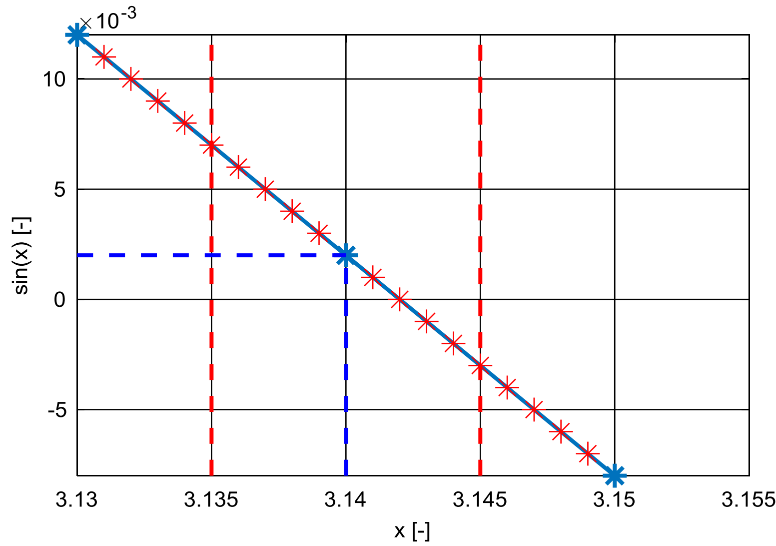

Figure 1-3: The Sine Function in the Given Floating-Point Number Format

The blue marks symbolize the correctly rounded results of the sine for all possible floating-point input arguments in the visualized interval. The red marks show possible numbers in the interval of the result. First it can be seen that the correctly rounded result for the representative of π, equal to 3.14 for the given floating-point format, is 0.002 and not zero. In case of the sine being calculated for the sum of two angles, the intermediate rounding must be considered.

For two angles:

$$\alpha = 2.23 \; and \; \beta = 9.06 * 10^{-1} \tag{1-3}$$

With $\oplus$ being the floating-point sum with rounding, the result evaluated in the given floating-point format is:

$$\sin(\alpha \oplus \beta) = 0.02 * 10^{-1} \tag{1-4}$$

Moreover, if the sine was correctly rounded the correct result rounded to the floating-point format would be:

$$\sin(\alpha + \beta) = 0.06 * 10^{-1} \tag{1-5}$$

Another aspect is that the input values to the systems might have significant lower precision than the machine precision. Therefore, an elementary function with correct rounding might not bring any benefit in comparison to an implementation which does not provide correctness of the last few least significant bits (LSB). An example where a high precision in an avionics system is needed is the position information. According to the specification of the avionics data bus ARINC 429 [8] the longitude is transmitted using two labels. The first one consists of twenty significant bits coded as a fixed-point number and giving an absolute value with a limited resolution. The second one provides eleven significant bits also coded as fixed-point number. The value is added to the first value in order to achieve a higher overall precision. The range of the signal is ± 180 which leads to a weight of the LSB of:

$$\frac{180°}{2^{31}} = 8.38 * 10^{-8}° \tag{1-6}$$

As for fixed-point numbers the bound of the relative error decreases for the higher values. So, the maximal relative error for a value close to $180°$ is:

$$e^{rel} \leq \frac{8.38 * 10^{-8°}}{2 * 180°} \approx 2.33 + 10^{-10} \leq 2^{-32} \tag{1-7}$$

Therefore, the relative precision is about 8 bit higher than the precision provided by an IEEE 754 single precision number [3]. But also 21 bit lower than the one provided by an IEEE 754 double precision number. A correctly rounded elementary trigonometric function might not bring any benefit under these circumstances for the embedded system.

The reason why correctly rounded functions still are not wide spread, is that for corner cases the calculation of the result needs to be performed with a high precision. As these high precisions are not natively supported by the underlying hardware the performance of such calculations is not high. In order to still get a higher average performance, calculations with different precisions and rounding tests can be used. For systems without hard time deadlines such an approach might be feasible, but if hard timing deadlines need to be kept the worst case will apply, which might be even higher due to the rounding tests in such cases. Also, the number of different branches makes it difficult to get tight bounds for the WCET as shown in [9]. Therefore, for real-time systems such an approach is not feasible.

This section discussed the challenges of elementary functions in an embedded system and especially for a safety critical flight control systems. Next to a proper specification, guaranteed precisions and execution time bounds are the main challenges in such systems. As this challenges are not completely new the next section gives an overview over existing works addressing these topics.

1.2 State of the Art

For the elementary functions many implementations exist which are delivered with the compiler or are freely available on the internet [10]. These libraries normally are directly available as object code or provided as source code. A detailed design description is normally not provided along with the library, also the achieved accuracy is normally not documented. This makes a use of such libraries in safety critical applications difficult.

One of the most popular works on the approximation of elementary math functions is [11], which was written some years before the IEEE floating-point standard [4] was published. Therefore, the numbers presented in this work do not exploit the characteristics of the number formats of the IEEE 754 standard. Some of the principles, especially for range reduction algorithms, are still very useful and also applied in this thesis.

Next to an implementation in software there are several works [12–14], which propose implementation of the elementary functions in a dedicated hardware. These implementations often use table lookup and shift and add operations. These operations can be easily and efficiently implemented in hardware, but for an implementation in software these algorithms normally are not a good solution. An implementation in hardware could either be already available in the target processor or a dedicated field programmable gate array (FPGA) can be used. The currently used processors in the embedded domain [15–17] normally do not provide an implementation of the elementary functions. Therefore, a dedicated FPGA would be

needed. But as such a device would be subjected to the DO-254 [18] standard, such an implementation would lead to an additional certification effort.

For the implementation in software some hardware vendors also provide library implementations [19, 20]. The purpose of those variants is to support an efficient implementation on their dedicated hardware. Therefore, these are normally implemented using assembler, which limits the portability. Although, the referenced works present some proofs of dedicated functions, in general a detailed documentation of those libraries is not available. So such implementations must be re-engineered for the use in an airborne software.

In the past years application processors were extended with the feature for single instruction multiple data (SIMD) operations [21, 22]. These operations can also be used for an efficient implementation of elementary math functions [23]. The work presents an approach to implement the functions using the parallelism of the underlying hardware, but for the accuracy bounds no proofs are given. Also, embedded processors nowadays do not support the parallelism features with high enough bits to get an approximation with higher than single precision.

[24] also uses the parallelism available in the applicable controller. In difference to the previous work here are handwritten proofs for the precision contained. The precision of the proposed implementations is up to $\pm 1\, ulp$ in the single precision format. The number of implemented functions is restricted and the proposed sine and cosine implementation is also limited to a basic interval. No range reduction algorithm is proposed nor are the rounding effects of the range reduction considered in the overall result.

[25] presents a comparison of the WCET and resource demands of table and polynomial based implementations. This work is limited to the cosine function and the used processor is simple as it does not support cache. For the polynomial approximations Taylor series are used, which do not give the most efficient approximation. Also, for the calculation of the error bounds only approximation errors are considered, but rounding errors were neglected. However, this work already shows that, with a processor, which does not incorporate a floating-point unit (FPU), the polynomial approximations are really competitive. The processor used in this thesis provides a double precision FPU. It is expected that the FPU increases the benefit of polynomial approximations. Therefore, the thesis at hand concentrates on polynomial approximations.

An approach to optimize the runtime of libraries is also presented in [26]. Here, stochastic approaches are applied in order to optimize the runtime of the libraries given by Intel. This stochastic approach is not applicable for a safety critical system. But as the results show a six times higher timing performance, it might be worth to reduce the precision a few bits in order to speed up the execution time.

In [27] the authors present a library that is correctly rounded for double precision numbers. It includes all basic math functions and also provides proofs for the achieved precision. The proofs are a mixture of handwritten proofs and generated proofs by the tool Gappa [28]. The attempt is to transform all proofs into generated ones. Although, the library aims to achieve a high computational efficiency, the performance is not comparable to a dedicated library for the specific use case in an embedded system due to the correct rounding property for all input arguments. Especially the fact that the hard to round cases are handled in extra branches, the variance in the execution time makes the application in a hard real-time system unfeasible.

No work could be found that integrates the library development of elementary math functions in the overall system development. For the model-based development the current approach is that the mathematical basic functions are considered as external components on the target system and on the development host system. This might lead to different results for the execution on the target and on the development hosts.

In [29] the use of a tool to check numerical accuracy of floating-point calculation in safety critical avionic software is presented. A static analyzer based on the method of abstract interpretation is used. In order to solve the given problem, a special domain is introduced in the tool. The paper contains the analysis of the implementation of two trigonometric functions and a filter algorithm. The focus here is only laid on uncertainties out of the floating-point calculations. The aspect of how the approximations of the chosen algorithm plus the floating-point errors sum up to an overall error are addressed only in a side note. The validation of the overall error is performed by using special mathematical equalities with a constant expression on one side. Implementing the non-constant term of the equality in the tool returns values in a range. Subtracting the expected value of the bounds of this range gives the overall error of the implementation. This is only shown for an approximation of the sine function using a lookup table algorithm. In order to gain a meaningful result, each interpolation point is considered separately. The work does not focus on the integration of these analyses in the overall development process and the development of the algorithms is not in the scope.

This thesis covers the integration of the developed library functions into a hard real-time system. For hard real-time systems one of the keys for a high utilization of the available computation resources are tight WCET bounds. In conjunction of cache with frequently executed code patterns, like libraries, cache locking is a good method to get tight bounds. Therefore, also an overview over the different available cache locking strategies is given here.

The different approaches can be allocated to two main groups. First, there are static cache locking approaches. Static means that the content of the cache is loaded and locked to the cache during an initialization phase and kept unchanged during the whole runtime. In contrast to this are the dynamic approaches, were the locking is executed during the runtime. Therefore, the content of the locked cache will change during runtime. These methods add additional code during runtime, so it must be carefully analyzed that the locking actually brings a benefit.

Arnaud and Puaut propose a dynamic instruction cache locking in [30], which should lock frequently accessed memory areas to the cache. To get the frequently accessed memory locations, a profiling is executed. Inserting the lock instruction afterwards in the code would lead to a different memory layout. In order to omit this, they called the locking routine via a trap handler executed at a certain program counter value. The trap handler is invoked by the debug utilities of the central processing unit (CPU). For an actual system this might be a limitation as the available hardware break points are normally strongly limited. The timing model used for their analysis only considers the effect of cache. With this model their experimental results show that in comparison to a least recently used (LRU) replacement policy their algorithms often lead to a higher worst case cache rate. Only in some of the examples their algorithm performs a little bit better. Therefore, this approach brings a better predictability but not necessarily a better performance compared to a cached system.

In [31] an approach to dynamically lock the data cache is presented. During the compile time the parts, which shall be locked to the cache, are selected based on a static analysis. In the generated code the corresponding loading and locking instructions are then automatically

generated and added at the corresponding locations. Also, this approach decreases the performance in some cases with the benefit of a higher predictability.

For timing analysis the different tasks allocated to a CPU are often considered independently. In case of cached systems this assumption does not necessarily hold true. Therefore, [32] considers all tasks allocated to one CPU. By applying a dynamic cache locking during the task switch a high utilization of the overall system is achieved. In this work the complexity is introduced by the preemptive multitasking system. In order to decide if a memory segment shall be locked or not the amount of possible preemptions is considered. This will lead to the circumstance that parts, which are frequently called, are not necessarily locked in case they are potentially more often preempted.

As dynamic lock instructions always lead to additional code in case of nested loops, it is quite crucial where exactly the locking algorithm is applied. This is considered in [33]. In comparison to a static analysis the approach in [33] can reach a WCET improvement in the range of ten to forty percent.

For multi core systems some of the caches are shared between different cores. Therefore, in addition to the intrinsic uncertainty of a cache memory, the interference of the different cores must be considered. This fact makes the research on cache locking methods in multi core systems to an active area of research. [34] presents an overview over different locking and partitioning approaches for a shared cache in a multicore system. By applying the proposed methods the runtime could be improved up to 40 % in comparison to a system without cache. [35] is also addressing the problem of shared cache in a multi core. In order to evaluate, which parts shall be locked, first an instrumented version of the software is executed. During this execution the frequency of memory accesses is profiled on OS page level. In order to get the actual allocated addresses, the software gets executed a second time without instrumentation. Out of this profiling the “hot” pages with a high access frequency are extracted. A coloring scheme is then used to solve the interference between the different hot pages by also remapping the physical memory addresses. Next, a dynamic cache locking is implemented at task level, which locks the hot pages of the corresponding task. In the presented examples the number of hot pages is chosen such that approximately 80 % of all memory accesses are contained within the hot pages.

Simple algorithms for getting a static cache locking are presented in [36]. In the work two approaches are presented for getting a high overall CPU utilization and one for minimizing the interference between different tasks. For selecting the memory blocks which shall be locked to the cache the limitations of mapping memory to a cache are considered. In order to get a high CPU utilization, the blocks with high calling frequency are locked. The WCET path is only considered at the beginning of the allocation. The fact that cache locking actually might change the program flow with the highest execution time is not considered in this work.

The effect that the WCET path might change, by locking some parts to the cache, is considered in [37]. Here a static locking approach is presented which selects, based on the memory size and execution frequency, certain nodes of the WCET path for locking into the cache. After each lock, the essential parts of the WCET analysis are again executed, in order to take into account that the locking might have impact on the program path with the highest execution time. As the approach only repeats the necessary parts of the WCET analysis, the problem can be solved with a reasonable calculation effort.

To solve the problem that memory blocks cannot be freely allocated to cache memory, the authors of [38] are proposing an adoption of the compiler optimization. During compile time a tree of the application is build. The nodes with a high execution count are selected for locking into the cache. The compiler takes also care that the nodes are placed in a way that they also fit to the cache.

Also, the way to identify the blocks, which shall be locked, are quite different next to profiling and solving some problems by integer linear programming (ILP) in [39] an approach based on a generic algorithm is presented. Similar to other works, here the main complexity is driven by the preemptive multitasking.

Most of the presented approaches here lock the whole cache. But as shown in [40] a combination of partial cache locking with static analysis might lead to better results than a static lock of the complete cache. The paper contains two approaches. For the first a complete calculation of the cache state is necessary. This is not feasible for programs with complex data flow. Therefore, also a heuristic approach is presented. For the heuristic approach the result of the static analysis is used to calculate the gain locking of different memory blocks. Next, the WCET analysis is repeated considering the block with the highest gain as locked to the cache. The gain is defined as the benefit in execution time minus the cost for locking the block. This is repeated as long as the analysis shows a smaller WCET and the gain of the estimation remains positive. As the complete WCET analysis is repeated after locking each block this approach is quite computation intensive. Overall the average improvement of the WCET is approximately 23 %.

The sections shows that for the state of the art the development of elementary math function is normally not integrated into an overall development process. Therefore, effects like caching and runtime optimizations due to restricted input ranges are not considered for the library development. Many available libraries lack of a proved precision, especially those which does not provide a precision up to the last bit. These topics shall be addressed by this thesis. The objectives deducted from these circumstances are presented in the next section.

1.3 Objectives

The goal of this thesis is to provide the implementation and integration of elementary math functions necessary for a modern flight control system. As shown in the previous subsections, due to the missing precision definition there is a gap in the specification of basic math function. Therefore, this thesis evaluates a meaningful precision for the elementary math functions. Meaningful in this context means that the precision should be a good tradeoff between the achieved precision and the necessary computational power. In order to get an efficient implementation, the algorithms are restricted to an input range, which is actually necessary for the embedded system. As some corner cases related to huge inputs can lead to the necessity of a complex range reduction with a high computational effort. In order to get a high performance, the benefits of a double precision FPU shall be incorporated. For current embedded platforms double precision FPUs are still not standard [17, 41] but as this is subject to change [42], this technology is already considered here. Another goal is that the developed implementations are easily portable to other targets. The portability is also important as the functions are integrated to an application, which is developed using a model-based approach. In order to enable a full integration of the developed basic math functions into the model development process the functions must be portable between the target system and the

development host system. The portability aspect is fully covered by this thesis. For integration aspects, which need to be considered for the model-based development environment, an overview is given.

For safety critical systems V&V is an important part. For that reason, aspects to ease these activities are considered here. Due to the high number of available numbers, testing of floating-point algorithms could never cover all cases. Therefore, the precision of the basic functionalities is verified by formal proofs. In order to minimize errors during these steps, automatic proof generation based on the implementation is applied. Further, V&V related considerations presented here are not limited to the pure library development but also consider integration aspects. For a runtime error analysis special stubs are provided in order to abstract the behavior of the library in such a way that a meaningful result is achieved. This result is also used to show that the assumption of the restricted range is valid for the given use case. In case the stub introduces additional assumptions about the correlation of input and output of the libraries the formal proofs are extended by these correlations. Last but not least the WCET aspect is considered. For that reason, the structure of the libraries is designed in a way that the control flow can be easily extracted and no additional annotations are necessary during the WCET analysis. As libraries are quite frequently called they are potential candidates for hot spots in the execution time and it is worth to also optimize their performance. This is done via two different approaches. During the design the algorithms with the best precision execution time ratio are selected. Afterwards, during the integration also the overall WCET is considered especially if the timing can be improved by preloading and locking some functions to the cache.

These objectives lead to an extension of the state of the art. The contributions to the extension are documented in detail in the next section.

1.4 Contributions

This thesis demonstrates a novel development and integration of elementary mathematical functions into a modern flight control software. The development is motivated by the guidance for airborne software [2]. So it should be possible to transfer the work to a certification project. This thesis aims at easing the integration of basic mathematical functions into a safety critical software, especially in a model-based development. The overall goal is to provide elementary math functions, which fulfill bounds for the precision and execution time. As the functions are constructed in a way that they can be used early in the development process, a smooth workflow is ensured. Therefore, the state of the art is extended by this work especially in the following points:

- **Coupling development of the basic math functions with the static runtime error analysis:** During the development only a restricted input range is considered in order to get to an efficient implementation. In order to proof the validity of these input ranges, a static runtime error analysis is executed. For the methods applied during the runtime error analysis it is hard to get a precise result for the evaluation of the approximation polynomials. To omit the issue concurrent with the development of the libraries stubs were developed, in order to abstract the input output behavior such that an overall precise result is achieved. The validity of the assumptions implemented in the stubs is shown along with the proofs of the precision of the function.

- **Considering model-based development aspects for the development of elementary math functions:** Due to restrictions of the hardware the elementary math functions used during the model-based development normally differ. This potentially could lead to errors, which are not observable at model level. In order to overcome this issue, the elementary math functions are implemented such that they are fully portable between the development host and the target computer. Since Intel has introduced a native support of the fused multiply add instruction since the 4th generation of the Core processor family [43], the use of the fused multiply add is also possible. This leads to an efficient implementation on both sides.
- **Accounting for timing issues during the development:** In order to enable the use in a hard real-time system, the presented approach considers several aspects. First the selection of the approximation algorithm is not purely based on the achieved precision but also the WCET is considered in this step. The architecture and design of the functions are selected such that the control flow can be well extracted with the method of the abstract interpretation. As this method is state of the art for WCET analysis [9, 44] this eases the overall development process. Here special care is taken on loops as it is essential that a correct loop bound can be evaluated during the WCET analysis. Finally, also the WCET in the integrated system is considered and here especially how it can be influenced by cache locking the elementary functions.

1.5 Outline

The remaining part of this thesis is outlined in the following: It starts with an introduction to the basic theoretical background in chapter two. The aim is to introduce the basis of floating-point arithmetic and common pit falls, which are relevant for the library development. Also, the principles of the WCET analysis are introduced as these make up an important part of this work. In chapter three a description of the environment, which is used to demonstrate the integration aspects, is given. It contains a short overview over the application software and the used hardware platform. The description of the actual development of the elementary functions starts in chapter four. First, an overview over the development process is given. Also, some architectural considerations and common patterns are described here. Beginning with chapter five, the different functions are presented. Chapter five deals with the trigonometric functions. These are the most relevant for the flight control algorithms. As all follow a similar implementation pattern, they are grouped together in one chapter. The development of the sine and cosine is described in most details as many patterns described here can be transferred to the other trigonometric functions. As the implementations for the square root and the exponential functions strongly differ from the others, their description follows in chapter six and seven. Chapters eight and nine cover V&V activities. First, the runtime error analysis is presented, which was executed in order to show that the assumptions on the implementation range are valid. Then, the WCET of the overall system under certain conditions is considered in chapter nine. Finally, this thesis concludes with chapter ten, where a conclusion and potential fields of future development are given.

2 Fundamentals

In this chapter an introduction to the necessary theoretical background is given.

2.1 Nomenclature

One of the most difficult parts in the context of floating-point approximation is not to mix floating-point numbers and operations with their real representative. In order to omit these pitfalls here a clear nomenclature is presented, which is used within this thesis. Most of the symbols are well established in literature, but also some special definitions are made here. Also, expressions for the errors are defined as these are commonly used in this thesis.

In order to be able to differentiate between real numbers and their representative in the floating-point number format a "~" is placed over the corresponding symbol. So, the floating-point representative of a is expressed as $\tilde{a}$. In case no deviation is stated in this thesis normally the IEEE 754 double representative is used in this thesis. Also, for the basic operations a differentiation between the mathematically correct and the operations with rounding must be possible. Consequently, in case a floating-point operation shall be shown the corresponding operator is surrounded by a circle. For the basic operations sum, subtraction, multiplication, and division the following symbols are used for the floating-point operations:

- $\oplus$ Floating-point sum
- $\ominus$ Floating-point subtract
- $\otimes$ Floating-point multiplication
- $\oslash$ Floating-point division

As numbers in a digital system are normally stored and processed with a binary base the introduction of a binary based exponential notation makes sense. Similar to the e in the decimal base system a b is introduced. So the expression $x\ b\ y$ represents: $x \cdot 2^{y}$.

In order to easily express rounding operations to integer numbers the following notation is used:

- $\lfloor x \rceil$ Round to nearest integer of x
- $\lfloor x \rfloor$ Round to largest integer smaller or equal than x (Floor Function)
- $\lceil x \rceil$ Round to smallest integer greater or equal than x (Ceil Function)

In case real functions are calculated by an approximation in a certain floating-point system several different errors occur. In order to get correct results, special care needs to be taken to not mess these different errors up. Therefore, here a special notation is introduced. The index is used to express the source of the error. First an error is made by approximating the function even if this approximation would be calculated with infinite precision. The approximation is expressed by the index apr, meaning that the approximation error is e_{apr}. In case a single or a sequence of instructions is executed using a common floating-point format a rounding error is added. In order to express the pure rounding error, the index rnd is used. Therefore, the rounding error is expressed as e_{rnd}. The approximation error and the rounding error sum up to a complete error expressed by the index tot as e_{tot}. Beside the source of error, errors also can have a different reference. This is expressed by a corresponding superscript. An absolute

error can be calculated by simply subtracting the approximated or rounded result from the correct result. This is indicated by abs so the error is expressed as e^{abs}. If this error is brought into a relation with the correct result, by dividing the absolute error by the correct result, a relative error is gained. This relative relation is indicated by the subscript rel, so the relative error is e^{rel}. Indices and superscripts of the errors can be freely combined to express the source and reference of the error. The absolute error is calculated as:

$$e_{apr}^{abs} = x_{apr} - x \tag{2-1}$$

The relative error is defined as:

$$e_{apr}^{rel} = \left| \frac{x_{apr} - x}{x} \right| \tag{2-2}$$

2.2 Floating-Point Arithmetic

In general, a floating-point number consists of an exponent, a significand, and a base. In order to get the represented number, the significand is multiplied by the base times the exponent. Figure 2-1 shows the different parts of a floating-point number with a decimal and a binary base.

$$\underbrace{2.25}_{\text{Significand}} * \underbrace{10}_{\text{Base}}^{1\,\text{Exponent}}$$

$$\underbrace{1.01101}_{\text{Significand}} * \underbrace{2}_{\text{Base}}^{4\,\text{Exponent}}$$

Figure 2-1: 22.5 Expressed as Floating-Point Number with Decimal and Binary Base

As the reversion, from a real number to a floating-point number, is not unique, an additional criterion is introduced. In general, a floating-point number is called normalized when the significand is in the range $[1, base[$. In order to store a floating-point number in digital memory, a dedicated number of bits are reserved for the exponent and the significand. This limits the amount of the available numbers. The number of bits and the definition how the exponent is coded leads to a minimal exponent. As normally the gap between the smallest normalized floating-point number and zero is quite huge de-normalized floating-point numbers are introduced. Also, with normalized numbers it would not be possible to represent zero. For de-normalized numbers the condition that the significand is in the range of $[1, base[$ does no longer apply, instead it is replaced by the condition that the range of the significand is $[0,1[$. In order to identify de-normalized numbers, the smallest exponent possible indicates a de-normalized coding. Next to the de-normalized and normalized numbers the IEEE 754 also introduces the special values infinity (INF) and not a number (NaN) [3]. To indicate such a value the biggest value of the exponent is reserved. For floating-point numbers with the biggest exponent, the coding of the significand is used to differentiate between INF and NaN. In case all bits in the significand are cleared INF is expressed. When at least one bit in the significand is set then NaN is represented. Processors normally use a binary based number system, so the use of the base two for floating-point numbers seems natural. In case of this selection in the significand one bit can be saved. For normalized numbers the significand is at least one but also smaller than the base, this corresponds to the first bit always being set. As it would be senseless to store information, which is anyway clear, this bit is not stored explicitly. This is

also known as hidden bit [45]. In order to also represent negative an additional bit is stored which indicates the sign.

Before the introduction of the IEEE 754 [4] many different specific implementations of floating-point implementation existed [46]. But after the standard was published it was broadly accepted, so that today almost all platforms make use of the definition of this standard. This increased the portability of floating-point instruction between different hardware platforms. The scope of the standard includes floating-point formats, rounding modes, coding of special values, basic operations, and exception handling. The standard does not define how the actual mapping to the memory is implemented. For an easier readability this thesis uses a big endianness representation in case some number is visualized as single bits.

The standard defines five different floating-point formats, three with a binary base and two with a decimal base. For this thesis actually only two are relevant. The first one is binary32, which is also known as single or float in many program languages [47]. As shown in Table 2-1 the total size of the number is 32 bits. One bit is reserved for the sign, eight for the exponent and the remaining 23 for the significand.

31	30								23	22																					0
Sign	Exponent									Significand																					

Table 2-1: Bit Representation of a Single Number

As the IEEE 754 uses a hidden bit for the binary-based numbers, the accrual precision of a normalized single number is 2^{-24}. The exponent is coded as a positive number. In order to get also numbers smaller than one and bigger than minus one, the actual exponent is achieved by subtracting a bias of 127 from the bit representation. This leads to a range of $[-126{,}127]$ for the exponent of normalized numbers. The lowest value of the exponent is reserved for the de-normalized numbers. Zero is also part of the de-normalized numbers and is represented by the lowest exponent and the significand being zero. For zero the sign bit can be either set or be cleared. Therefore, it is possible to differentiate between plus and minus zero. For example, this is used when a negative number underflows, then the sign can be kept. As a check of equality of plus and minus zero returns zero programmers normally do not have to take care about this specialty. As stated before, the biggest exponent is reserved in order to represent INF and NaN. The single precision normally is sufficient to represent typical values in a flight control system. Only for the position information the resolution can get too low. Like for every floating-point number the absolute resolution decreases with increasing values. Therefore, it depends on the actual position if a single number is precise enough. In the worst case, on the equator at 180° longitude, the absolute position error goes up to the range between one and two meters in case a single number is used. As this might not be sufficient enough for precise flight maneuvers for the position information a higher resolution is used. This information is stored and processed using the IEEE 754 binary64 format, which is also known as double in many programming languages. As shown in Table 2-2 the total size of the number is 64 bits. One bit is reserved for the sign, eleven for the exponent and the remaining 52 for the significand.

63	62										52	51																			
Sign	Exponent											Significand																			
																															0
Significand																															

Table 2-2: Bit Representation of a Double Number

The coding of the number is similar to the single number. The bias of the exponent is 1023 which leads to an exponent range of $[-1022,1023]$ for normalized numbers. The meaning of the biggest and the lowest exponent are similar to the single number. Again, the hidden bit is applied to the significand. This leads to a precision of 2^{-53}.

As the values of all representable floating-point numbers are limited, floating-point numbers normally are subject to rounding. Therefore, IEEE 754 also takes care of the rounding behavior. Four different rounding modes are defined:

- Rounding to nearest
- Rounding toward positive
- Rounding toward negative
- Rounding toward zero

Rounding towards nearest is the default and most commonly used rounding mode. Therefore, all considerations in this thesis are limited to that particular rounding mode. In case of round to nearest the default behavior for ties is the round TiesToEven [3]. This means that for ties the significand where the least significant digit is even is selected. So for example the number 3.145 would be rounded down to 3.14 and the number 3.155 would be rounded up. In this thesis rounding ties to even mode is always used.

Beside other operations the IEEE 754 defines that the following instructions have to be calculated correctly rounded:

- Addition
- Subtraction
- Multiplication
- Division
- Square Root
- Fused Multiply Add (FMA)

Correctly rounded in the context of the IEEE 754 means that the overall result shall have the same value as in the case that each intermediate instruction is executed with infinite precision. After the calculation, then a final rounding according to selected rounding mode is executed in case this is necessary to fit the number in the destination format.

Although the FPU of cores normally supports the IEEE 754, not necessarily the complete standard is supported. In those cases, additional software is necessary in order to get full support. For example, the target used in this thesis does not support a square root calculation in hardware [15]. Also refer to chapter 6 for more details on this topic.

The fused multiply add instruction calculates $(x * y) + z$ without an intermediate rounding and the precision can be increased with this instruction in comparison to a sequential multiplication and addition. For some processors the latency of this instruction is also lower than for the two sequential operations [15]. But actually, this is a bit platform dependent as on some platforms the use of sequential instructions with extended double precision might be faster as the pipeline is better used in those cases [48]. But as the extended formats normally are not available on the embedded target, the implementations in this paper make use of the fused multiply add.

As errors in the context of floating-point numbers are normally quite low, it is quite inconvenient to express errors as real numbers. In addition when the binary base it used, these errors are

in addition normally also a number to the power of two. In order to ease dealing with errors the unit in the last place (ulp) was introduced. A casual definition, the ulp is the distance between two consecutive floating-point numbers. There is no generally valid definition of the ulp [49]. In general all the definitions do not vary too much in case the number is not close to a power of the base. In this thesis the author sticks to the ulp definition according to Goldberg with the extension to reals [49] as this is the most commonly used. With p being the precision of the floating-point number and e being the exponent, the ulp is defined as follows:

"If $x \in [\beta^e, \beta^{e+1}[$ then $ulp(x) = \beta^{max(e,e_{min})-p+1}$."

In general, all operations with floating-point numbers are subject to rounding, but there are also some operations that can be calculated exactly. For example, a multiplication of the base with any floating-point number does not lead to any rounding as long as the result still is a normalized number. The design of elementary functions can benefit from those operations without any rounding, as they keep the overall error low. One of the most commonly used lemmas in this context is Sterbenz lemma [50]. It is defined as follows:

"In a floating-point system with correct rounding and subnormal number; if x and y are floating-point umbers such that

$$\frac{x}{2} \leq y \leq 2x \tag{2-3}$$

Then $x - y$ is a floating-point number, which implies that it will be computed exactly, with any rounding function."

This lemma must be taken with care, as for floating-point numbers there is also the effect of catastrophic cancelation, which says that when two operands, which are subject to rounding, are subtracted, then the floating-point result might have an error of several ulps. First this seems to be contradicting to Sterbenz lemma, but the Sterbenz lemma is valid as it takes floating-point numbers as input, where the actual subtraction in the case of catastrophic cancelation does not introduce an error, it only reviles the previous rounding error. In [45] an example with the quadratic formula can be found where an error of 70 ulp occurs by the calculation of the radicand. As elementary functions normally deal with real numbers and not with floating-point values, special care of the correct use of Sterbenz lemma must be taken.

2.3 Worst Case Execution Time

In order to prove the stability of control algorithms the overall delay of the control system, from the sensor to the actuator, must be known. As the actual calculation of the control algorithm is one of the contributors to this delay, it is mandatory that this time is known. Figure 2-2, which is shown in a similar way in [44], gives an example diagram for the execution time of an algorithm in a computer system.

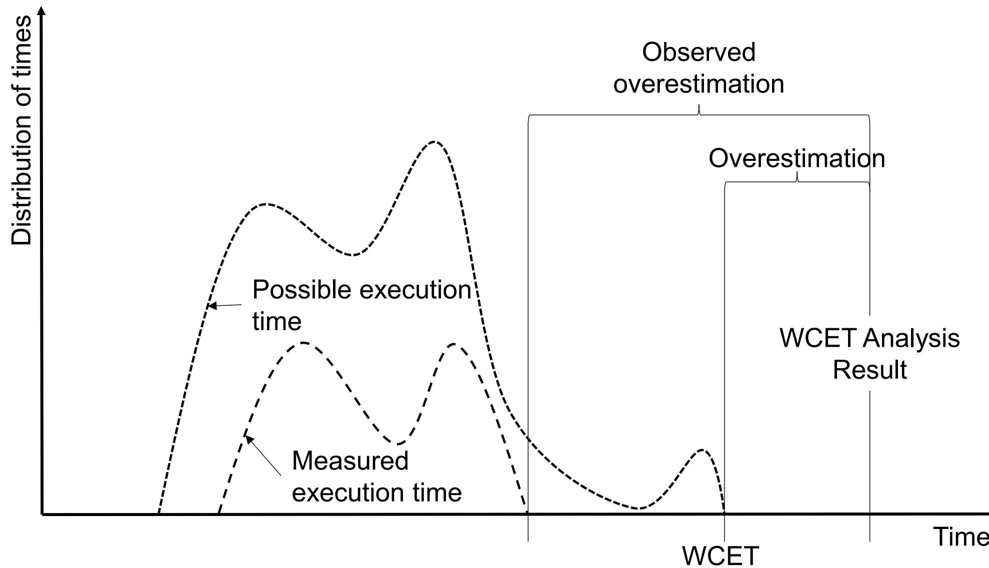

Figure 2-2: Measured vs. Possible Execution Times vs. WCET Analysis Result

The calculation time is not constant and might vary dependent on the current input and the history of inputs. The longest possible execution time is the time, which is of interest for the verification of the stability of the control algorithms. In programs with a complex control flow it is not possible to determine this time by pure measurements. Therefore, certain rules are used to calculate a WCET. As the used rules normally lead to an overestimation, the analyzed WCET normally is bigger and not equal to the highest possible execution time. The goal is to keep this overestimation as small as possible. As the WCET is an important number to show the stability of the overall system also the DO-178 [2] explicitly requires that the WCET of the software is determined for design assurance level (DAL) A to C. How the WCET shall be determined is not specified. For modern CPU architectures the evaluation of the WCET is quite complex. As these CPU architectures feature components like pipelines and caches, which make execution time depending on the context and the previous executed instructions. For example, when an instruction is cached the execution time might be 20 times lower in comparison to when it is not cached [51]. Systems like [17, 41, 52] used in the embedded domain feature both pipelines and caches for data and instructions. Therefore, in order to get a tight bound for the WCET the complete integrated software needs to be considered for the determination of the execution time. In case the WCET should be determined in an end-to-end measurement, all different branch combinations in the software must be triggered. For software with complex control flow this is normally not possible. In order to ease this process a static analysis process is the standard approach for the cases where a sound WCET shall be calculated [9]. In this thesis the commercial tool aiT is used for the WCET analysis. In the following the execution sequence of the tool aiT is described while other tools often have a comparable workflow [44]. A more detailed description can be found in [53]. As first step of the analysis the control flow is extracted. Input for this is the executable as it also contains potential compiler optimizations. In order to determine number of loop iterations or infeasible path and the access of memory areas a value analysis is executed using the approach of abstract interpretation. For cases where the control flow problem cannot be solved by the method of the abstract interpretation, user annotation can be introduced to give further information. Afterwards the execution times of the basic blocs shall be determined. As the cache and pipelines have an important impact on the timing, their state has to be considered. The analyzer here also uses the approach of abstract interpretation to calculate their states for the

different control flows. With the cache and pipeline states the timing of the basic blocks can be calculated. As last step a path analysis is executed. For this process ILP or similar methods are used to calculate the actual WCET path out of the control flow path to which the timing of the basic blocks was added.

As this thesis also analyzes the impact of the cache and its configuration on the analyzed WCET, some basics of the cache are explained here. Cache is a fast memory close to the core, which is filled dependent on the program execution. The purpose of this memory is to lower the latency of fetching data from the memory. As for main memory accesses several busses, cross bars, or even memory controllers must be accessed the fetching of instructions or data can lead to a huge delay. Also, for many modern architectures the frequency of the memory is lower than the one of the core. Therefore, it is possible that the calculation stalls if not enough instructions or data are available in the core. The cache with the low access latency shall help to omit CPU stalls.

Today the most common organization of cache memory is the n-way set associative cache. The cache organization can be visualized as a matrix.

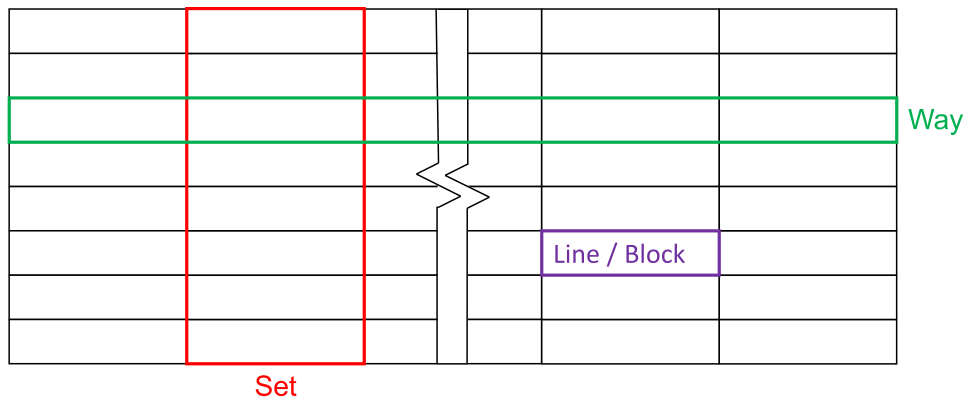

Figure 2-3: Cache Structure

The height of the matrix are the n-ways and the width are the number of sets. One set consists out of the complete column of a matrix. The entries in one set are called blocks or lines. Each memory address is allocated to a certain set, but within this set, it can be allocated to any block. Therefore, the actual way in which the data are loaded depends on the context [51]. In order to select a way several approaches exist. First, all cache lines are filled, but, in case all are already filled and additional data is loaded, a replacement policy defines, which line should be replaced. Three commonly used ones are first in-first out (FIFO), least recently used (LRU), and pseudo least recently used (PRLU) [51, 54]. The first in first out is quite straightforward the block that was cached first is also replaced first. This principle should be familiar to every programmer. The least recently used removes the block that was not accessed for the longest time. So in contrast to the FIFO policy, here also accesses to already cached locations are considered. The pseudo least recently used policy also tries to eliminate the least recently used, but instead of comparing each entry, a binary tree is built which always points to the line to replace. In case of an access, each element of the tree leading to the access is flipped. Especially for caches with many ways, this implementation is more efficient compared to the least recently used policy. Beside the replacement policy CPUs normally also provide some possibilities that the user can manipulate the cache content.

Some CPUs provide the possibility to lock the cache entirely or parts of this. Locking in this context means that the applicable part of the cache is not subjected to the replacement policy and kept in the cache. The granularity of cache locking depends on the platform. Some CPUs provide the possibility to lock certain lines, others only to lock complete ways [15, 16]. As shown in the introduction cache locking can be used to minimize the task interference or getting a smaller WCET. Many of these approaches require that a known content is part of the cache. In order to achieve this special load, routines are necessary. In order to ensure that the locking routine is not loaded into the cache during execution this function is normally allocated to a not cache-able memory section. To load instructions and data into the cache, they can either be accessed or the CPU provides special instructions for this purpose.

3 Demonstration Environment Description

The goal of this thesis is to show the implementation and integration of the necessary elementary math functions for a flight control system. To show also aspects of the integration into a complete system, here an experimental flight control software is used. This software was developed in order to control a research aircraft of the Institute of Flight System Dynamics (FSD) of the Technical University of Munich (TUM). This aircraft is a twin-engine four seat aircraft. Figure 3-1 shows a photograph of the research platform. The flight-control system of this aircraft has been modified such that an experimental fly-by-wire system has been installed in parallel to the mechanical flight control system. Furthermore, also additional sensors are equipped to enable closed-loop automatically controlled flight [55]. The demonstrated approach can be used with different software projects. So, this environment only has a demonstration character.

Figure 3-1: Research Aircraft from TUM FSD

3.1 Flight Control Computer Hardware Overview

This section gives an overview of the hardware, which is used for execution of the presented software. The system was developed under consideration of the applicable certification regulations and keeping in mind the goal to ease validation and verification of the system. In order to fulfill these requirements a multi CPU approach was selected. As shown in the diagram in Figure 3-2 the system consists of two Cortex-M3 processors and one MPC8349 CPU [56].

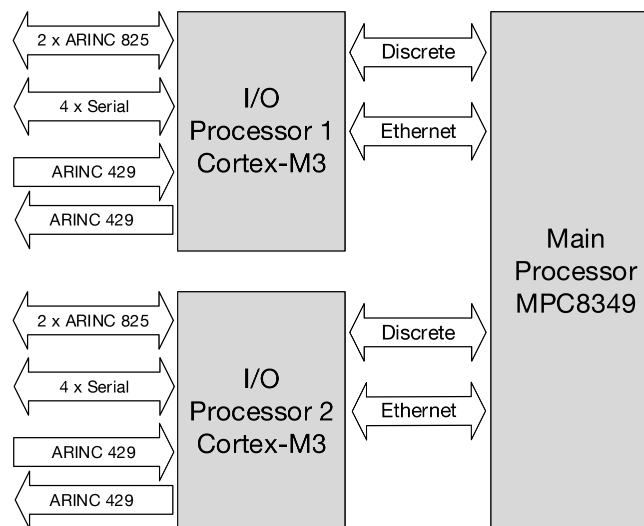

Figure 3-2: Flight Control Computer Structure [57]

The two Cortex-M3 processors are connected to the external interfaces. Therefore, they are also called input / output (I/O) processors in the following. The MPC8349 is responsible for the actual calculations and is also the master in the communication to the I/O processor. Therefore, it is also called main processor. The figure shows the digital external communication interfaces. In total eight serial, four ARINC 825, and two bidirectional ARINC 429 interfaces are available. Beside these also some discrete I/Os and analog inputs are available, which are not displayed in the figure. All interfaces are equally distributed over the two I/O processors. The benefit of using a separate microcontroller for the external communication is that the handling of the external asynchronous interrupts due to communication does not disturb the execution of the application software. This breaks down the complexity of the timing on the main processor and the computational power of the CPU can be better utilized for the actual calculation.

To make the communication data available on the main processor, a full-duplex Ethernet communication is part of each I/O processor. As shown in the diagram to each I/O a separate Ethernet interface exists. Therefore, the communication is point to point and no collisions can occur on the physical line. To enable the main processor to request data from the I/O processor, a discrete line exists between the two CPUs in addition to the Ethernet. So it is possible that the main processor can request data by a simple discrete signal and must not send a whole Ethernet frame for data requests.

The MPC8349 is a 32bit PowerPC based system built on an e300c1 core [52]. The used processer is operated at a core frequency of 533 Mhz. In order to minimize the memory access latency, the processor is equipped with both data and instruction cache. Both caches are structured in the same way: They offer eight ways and 128 sets. Each cache block has a size of eight words (32 byte) [15]. This leads to an overall cache size of 32 KB. The user can enable or disable the cache. In addition to that a cache locking is possible. Either the full cache or certain ways can be locked by the user. The locked ways always start with way zero and increase in a continuous way. Locking on block / line level is not available. Next to the normal integer processing unit the core also provides a double precision FPU. The floating-point unit is compliant to the IEEE 754 but in order to fully cover the standard software support is needed [15]. The floating-point unit only supports basic instructions and does not natively support the square root function, which is part of the IEEE 754 in both editions [3, 4]. In order to store the constants and the program in the system, the main processor has 16 MB of flash attached. For the execution 256 MB of DDR SDRAM are available [58].

This thesis presents only software executed on the main processor. The software running on the I/O processor is not in the scope of this thesis.

3.2 Application Software Overview

The implemented functions allow a complete automatic flight of the aircraft. In order to enable an easy modification or an adaption to other platforms, the system is split into several modules. An overview over the different modules is shown in Figure 3-3.

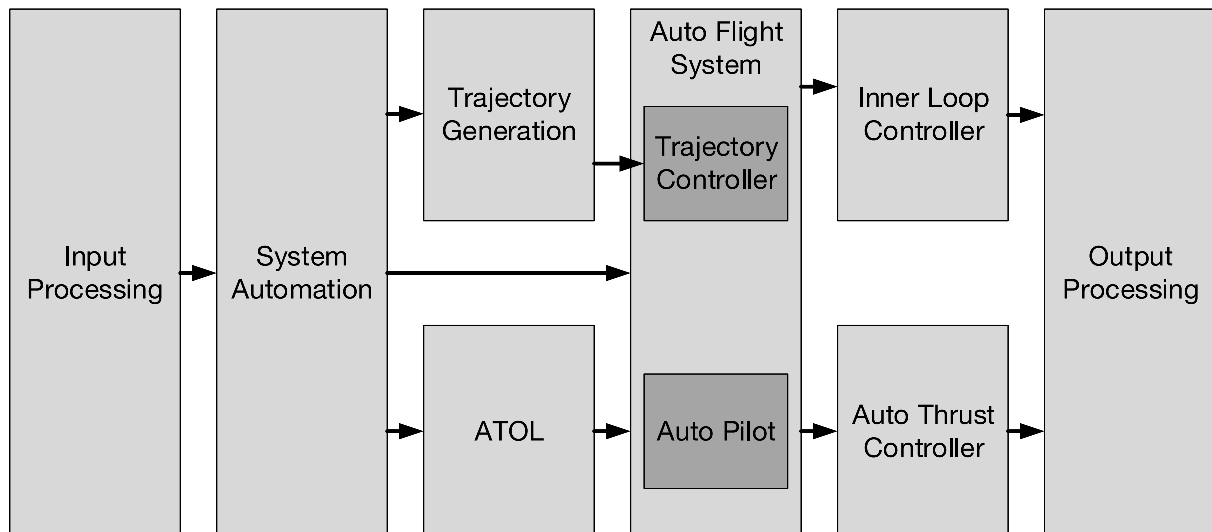

Figure 3-3: Flight Control Modules Overview [57]

First, all input signals are evaluated in the input processing system. Based on the available inputs and the selected operation modes, the system automation triggers the corresponding sub functionalities. A detailed description can be found in [59]. Besides system states such as stand-by, the system automation can trigger three different nominal high level operating modes for automatic flight. The first mode is a full three dimensional trajectory flight. The trajectory necessary for this mode is generated by the trajectory-generation module. Based on a given waypoint list, this system generates an achievable flight path and calculates the current deviation of the aircraft from this path [60, 61]. The second mode is automated takeoff and landing (ATOL). These critical flight phases are handled by a separate model. This model includes a separate trajectory generation and a mode logic specially designed for this flight phase [62, 63]. As a third alternative, an auto flight mode can be triggered. Here, operations, which are provided by an autopilot, like desired heading or altitude hold, can be commanded. More details on the modes and their implementation can be found in [64]. The auto flight module also contains a special trajectory controller, which reduces the deviation between the aircraft and the desired trajectory to zero [65]. The output of the auto flight module then forwards the commands as load factors to the inner loop controller and auto thrust controller. In the auto thrust controller the actual thrust setting is calculated with the load factor in direction of the flight path as input. The inner loop transforms the load factor normal to the flight path into deflections of the control surfaces [66]. All outputs are processed by a separate module, which then forwards all values to the applicable objects, specific for the target platform. All the functions described in this paragraph are implemented using a model-based approach in Simulink. The model-based development process is addressed in the next subsection.

3.3 Model-Based Development Approach

The main functionality of the flight control software at hand is developed using a model-based development approach utilizing The Mathworks MATLAB/Simulink. Here, only a short overview of the development process is given. A more detailed description can be found in [67]. In context of safety critical airborne software, the model-based approach brings the benefit that the developed model serves as software architecture and as low level requirements. The software architecture is necessary for all DAL levels. The software low level requirements are mandatory for DAL A to C by the DO-178 [2]. The next step after the software design is the

coding of the actual software. For a model-based design developed with Simulink this step can be automated by the Embedded Coder. For DAL A to C a review of the code is mandatory afterwards. In case the software architecture and low level design is captured in a formal language like model-based design, this step can either be completely omitted by the use of a qualified code generation tool. Alternatively, this step can also be automated by a checker tool, which then also needs a qualification. In the Simulink ecosystem the second step is chosen. Here, the Simulink Code Inspector (SLCI) is provided, which translates both the generated code and the input model to a special internal representation. Afterwards, these representations are checked for equality. To make this approach feasible, some limitations are added to the model. These are mainly a restriction to a subset of all available blocks and their settings, but also the ways how subsystems can be integrated is slightly limited. For the development an iterative approach is applied, which utilizes a lot of simulation on model level [68]. Therefore, the way to integrate different modules was selected such that a high performance in the model simulation is achieved. This leads to the effect that the method of referenced models is not heavily used. In the generated C source code this leads to quite huge functions, which becomes important in the later considerations.

The complete model, with all referenced models, is coded as one completely integrated system. Therefore, for integrating the model on the target, the surrounding C source code only needs to call two functions. First, the generated initialization function after the target comes out of a reset and the step function in a cyclic context [69]. The classically developed part of the source code needs to ensure that the step function is called with the same frequency as selected in the model configuration. Beside the call of these two functions the input data, like sensor data, must be provided to the Simulink model and the output data of the model, like the actuator commands, must be forwarded to the actual hardware interfaces. In order to achieve this, the interface control documents (ICD) have been extended by the information, which bus signal belongs to which Simulink signal. So as a convenient way to get this information to the source code, a special coder has been programmed. This coder reads the information out of the ICDs and generates the functions to extract the data out of the interface messages to the Simulink input data structures. As the bandwidth of communication interfaces is normally limited the data is often specially coded as some fixed-point value on the bus. In order to incorporate this behavior in the interface generator also a scale factor and an offset can be applied during the extraction. The output signals are processed vice versa. The Simulink bus structures are packed into the message structures and then forwarded to the applicable driver.

To enable a full capability signal monitoring inside of the model, in addition to the actual payload also information on the age of the data is added to each message. Therefore, an 8-bit unsigned integer is used. In order to indicate that a message was not received at all a value of 255 is used. Numbers in the range of 0 to 250 indicate how many cycles were executed since the last reception. The biggest value 250 is actually a special value as the count stops here. Therefore, this value has to be interpreted bigger or equal to a delay of 250 cycles. As the update of the Simulink data is actually only happening in advance to the execution of the model the age indication cannot get smaller than zero. The values from 251 to 254 are reserved for future use. The addition of this value is not explicitly stated in the ICD. This behavior is specified to the code generator. For that reason, the ICDs can be kept in-line with the documentation of hardware components. As the scheduling of the output data is part of the Simulink model, a similar method is necessary for the outputs. As the code which is handling the outputs is synchronous to Simulink, a simple Boolean value is sufficient here. If this value is true, the message will be forwarded to the corresponding output interface. Adding the scheduling of

messages to the model brings the benefit that effects due to discretization already can be checked during model simulation.

This is especially of importance as one of the goals of the model-based development process is that all relevant behaviors are correctly modeled. So, they can already be observed during the simulation on a development host. This brings the benefit of early error detection, as simulations normally are available earlier than the actual software tested on the target hardware. Early detection can lead to a significant reduction in development cost and time. One contribution of this thesis is the provision of elementary math functions that provide the same functionality on the target and the development host. In case these are integrated to the model, the confidence in the simulations is further increased.

3.4 Base Software

As explained in the previous chapter the source code generated out of Simulink does not cover all necessary aspects. Therefore, it is integrated in a framework-software, which is manually developed. In order to ensure an efficient implementation, validation and verification, one of the goals was to keep this part as simple as possible. The validation of the timing behavior is a big challenge for complex software projects. The use of a hardware, which uses context dependent features like cache and pipelining, also increases the effort for determining a valid WCET bound. To minimize the complexity of this problem, a simple scheduling strategy is used. The complete functional implementation is integrated to one static scheduled task. In comparison to a preemptive scheduled system no interference between different tasks must be considered. This affects the consistency of input data as the data might be changing during one task is interrupted by another task. However, it is especially of interest with respect to timings that in a non-preemptive system the uncertainties in the cache memory are not further increased by preemptions happening during any time of execution. Another alternative solution to a preemptive system would be a system with different tasks, which are allocated to a fixed schedule. Here, an allocation of software parts to the different tasks must be performed in advance. As the software at hand is developed in an interactive approach, this allocation might need adaption quite frequently. In order to omit this additional effort, the single task approach was chosen. This is sufficient, as it can be shown that this approach is schedulable on the target hardware.

The different phases of one cyclic task are shown in Figure 3-4.

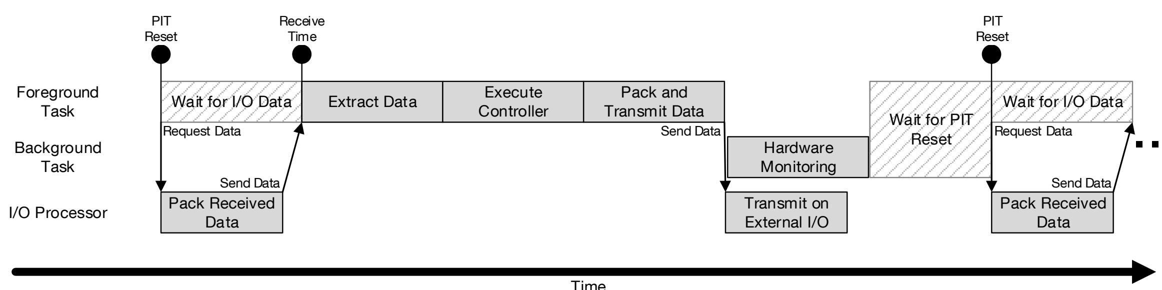

Figure 3-4: FCC SW Sequence Diagram [57]

The beginning of each time slice is triggered by the periodic interrupt timer (PIT). As first step, in every cycle the data are requested by the main processor from the I/O processor via a discrete signal. This signal triggers the transmission of all data received in the I/O processor

during the last cycle. The transmission is implemented with a custom Ethernet packet using the OSI layer two. After triggering the main processor waits for a fixed time for the reception of the Ethernet packages. The wait time is deducted of the worst case considerations of the I/O processors. This approach does not lead to the minimal average execution time, but as in a hard real-time system the WCET is of interest anyway, the average execution time is not of interest. The benefit of a fixed wait time also has the benefit that the main processor keeps on processing in case an I/O processor fails. Dependent on the allocation of the interfaces this can still lead to a robust system. When the Ethernet data are received the main processor begins with its actual processing. First all data are extracted out of the Ethernet package. In cases the messages on the physical bus are already identified by the hardware as for ARINC-825 or ARINC-429, the code generated by the interface generator is directly triggered during the extraction of the Ethernet package. In case the transmission protocol is implemented in software, like for serial interfaces, the data are first copied to a FIFO during the Ethernet extraction. After the Ethernet part is finished, complete messages are searched in the FIFOs. In case a new message is found again the code generated by the interface generator is triggered in order to forward the signals to the Simulink code. Afterwards the code generated out of Simulink is executed. Here, the actual flight control algorithms are calculated. The processing of the output data is similar to the input data. First all serial messages are checked whether they are updated. In case of an update, all signals of the message are converted to an ICD compliant byte stream, which is then also extended by the protocol artifacts like header, identifier, and checksum. This byte stream is then stored in a FIFO. Afterwards, the Ethernet packages are generated. For the serial data, the byte stream stored in the FIFO is forwarded to the Ethernet frame. For other data, where the actual message protocol is also part of the specification like ARINC-825, the routines generated by the interface generator are directly called by the Ethernet packing routine. If a complete Ethernet package is full or all data have been packed, the data are sent to the I/O processor. Here, the data are forwarded to the corresponding interface drivers after decoding the Ethernet frame.

In order to get the software to the target, the complete software of the main processer is compiled to a single binary using the complier CompCert [70, 71], which offers formally verified optimizations. As shown in a previous work [57], the use of the optimizations applied by this compiler can bring a benefit of the factor 2.7 for the WCET in comparison to a non-optimized version.

3.5 Software Statistics

This subsection concludes the environment description with some numbers of the software example at hand. The source code of this software has overall 157,967 source lines of code (SLOC). 144,372 SLOC are produced by the Embedded Coder out of the Simulink model. The rest is handwritten code and code generated by the custom interface generator. The handwritten code contains next to the C source code also 316 SLOC written in assembler. The assembler code contains the startup code and the cache locking and preloading functions. The compiled and linked binary of the software has a size of 1086 KB. The executable code has a size of 969 KB and the constant and initialized data consume 117 KB. During the runtime 98 KB are required for uninitialized data in the RAM and the maximum stack usage is close to 20 KB. The elementary math data are of special interest for this thesis. Therefore, in Table 3-1 an overview over the number of calls to these functions is given. The table differentiates between the single and the double version of the function. For both variants, the number of occurrences

in the source code (call sites) and the number of calls on the WCET path are listed. As the WCET path considers the actual control flow, the number of calls here can be smaller or greater than the number of call sites. For example, the count is greater in case the function is called in a loop or contained in a function, which is called several times. Less calls are on the WCET path in case some of the call sites are not part of this path as a parallel branch is taken or the code is infeasible. The reason why the table lists not applicable (N/A) for some single variants is, that the WCET path is based on a version with the custom math functions integrated. As explained in the following sections, for some functions providing a separate implementation for the single variant makes no sense and so the single and double version cannot be separated in the WCET analysis result. In those cases, all calls are listed in the corresponding column of the double variant. The calls on the WCET path for the arcus tangent also include the ones of the arcus tangent two, as this function is based on the arcus tangent.

Function	Single Variant		Double Variant	
	Call Sites	Calls on the WCET Path	Call Sites	Calls on the WCET Path
Sine	49	47	131	135
Cosine	58	66	118	103
Tangent	2	2	21	12
Arcus Sine	20	N/A	8	62
Arcus Cosine	0	N/A	28	36
Arcus Tangent	0	15	3	33
Arcus Tangent 2	15	15	34	28
Square Root	21	N/A	124	223
Power	6	N/A	788	509

Table 3-1: Usage of Library Functions

4 Library Development Process

This chapter is separated into two main parts. First an overview over the chosen development method is given. In the second part it is shown how the development steps can be mapped to required steps in a formal process, and what qualifications must be gained for the tools used here.

4.1 Workflow

Figure 4-1 gives an overview over the development process. The dotted arrows indicate a verification activity. The ones with a continuous lines an information / data flow.

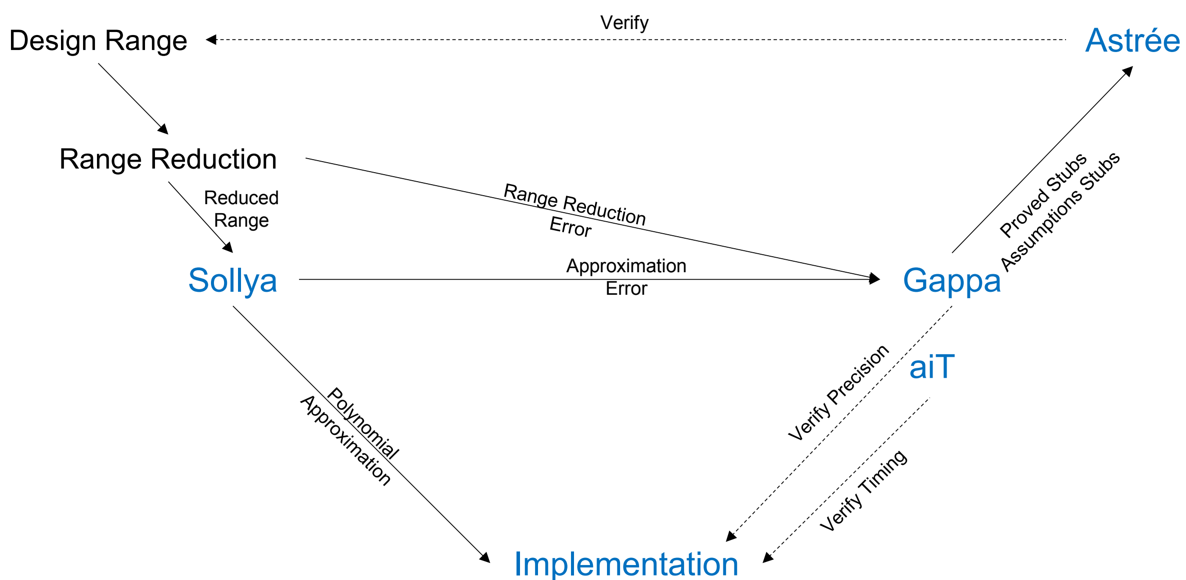

Figure 4-1: Workflow Overview

Before a function is developed, some boundary requirements are determined. Next to the mathematical expression, which shall be implemented, the following topics need to be considered:

- Input format and range
- Precision of the result
- Behavior in special cases

With exception of the exponential function presented in chapter 7 all inputs to the presented functions are floating-point numbers. Dependent on the desired precision the input is either chosen to be a single or double precision number. As the CPU at hand has a double precision FPU, for most instructions there is no performance penalty for the double variant in comparison to the single variant. For that reason, for the internal calculations all inputs are converted to double precision numbers. The internal use of double precision numbers enables an efficient implementation to reach the desired precision. For some functions the input range is the complete range defined for the double or single precision number. Others have a restricted design range. In order to decide on a design range, the knowledge of some internals of the functions is beneficial. Especially for the functions, which have an additive range reduction, the

complexity of the range reduction might increase with the range. So here the range normally is restricted. But for all functions the selected range is justified, based on the functional background. For example, the range of sine and cosine is set to $[-4\pi, 4\pi]$. As an angle is normally in the range of $\pm 2\pi$, the range is also sufficient if the addition of two angles is passed to the function. That all possible input arguments to the library functions are within the desired design range, is verified with a runtime error analysis of the integrated software. Next to the actual implementation of the function also special stubs are developed. During the runtime error analysis the calls to the elementary functions are replaced by calls to these stubs. The correctness of these stubs is shown with formal proofs.

For the precision no quantitative design goal was set prior to the design of the function. The reason therefore is that the work presented here also gives an overview which precision can be reached if double precision arithmetic is used. Due to the overhead, extended precision was omitted, as this is not natively supported by the hardware at hand. Therefore, basically the priory goal for the precision was a relative error as small as possible. The function examined in this thesis to the highest extent is the cosine. Here multiple polynomial grades with different precisions are discussed. As some of the expertise gained for this function is easily transferable to the other functions, for those only the cases to achieve the best precisions are considered. In addition, another version, which reaches a precision of 2^{-24}, is considered. The value 2^{-24} is chosen as it correlates to the relative error of a normalized single number. For a few functions it was necessary to define an expected precision during the algorithm design. For these cases a value of 2^{-52} was chosen as relative approximation error. The value is chosen as due to the rounding errors during the evaluation with double precision lower approximation errors normally bring no benefit.

Special cases are mainly applicable for inputs at the limit of the defined input range or outside of that range. The rational for those cases is motivated by two different reasons. The first is some intrinsic function of certain Simulink blocks in their SLCI specific setting. For example, the arcus sine and arcus cosine limit the input values outside of the range $[-1.0, 1.0]$ to the corresponding limit value. In order to get the same behavior in case the user decides to integrate the functions as legacy code, those limits are also included in the implementation of the applicable elementary function. As the limit values in most cases need a special handling inside of elementary function anyway, the implementation of the limitation inside of the function will be more efficient. The second reason for special handling might be due to numeric stability. So for example there are cases in which it is not possible to actually show that the square root is exactly zero. Therefore, the input to the square root might be a negative number close to zero. In order that no exception is raised in those cases also a zero is returned for negative numbers.

After these boundaries are defined, the actual design of the algorithm is started. Here different methods are applied. First, if a range reduction is necessary, an appropriate algorithm has to be chosen. Here normally a handbook method appropriate for the given input range is chosen. Also, the proof of the precision is normally executed by hand here. This method was chosen, as there are already a bunch of methods available with the corresponding proofs. This proof normally only needs small adaptions in order to fit the given use case. This is faster than transferring the complete proof in a script to compile an automated proof. For the functions, which are implemented in this thesis, only the trigonometric ones, presented in chapter 5, need a range reduction. The square root and the power function, which are also in the scope of this thesis, the designed algorithm is proven to work on the whole design range without a range

reduction. After the range reduction an approximation of the actual function on the full or reduced range must be designed. Here again the trigonometric functions differ from the rest of the other functions. For the power function and the square root numerical methods are chosen, which are described in detail in chapter 6 and 7. For the trigonometric functions mainly polynomials are used. In order to get a good approximation also considering the used precisions of the coefficients the tool Sollya [72] is used. Sollya is a tool for the development of polynomial approximation and for the calculation of infinity norms. It is dedicated for safe floating-point development. In order to ensure a safe computation, a high precision is used internally for the calculations. To determine a good approximation, different approaches of the approximation are evaluated. These vary in parameters like different polynomial degrees or a variation of the approximation ranges. In order to handle the different variants, the scripting capability of Sollya is used. To calculate the polynomial approximations normally the function `fpminimax` of Sollya is utilized. This function has the following input parameters:

- Function which should be approximated
- Grade of the polynomial approximation
- Precision of the coefficients
- Input range for which the approximation shall be developed

In order to get an overview over the different reached precisions, the used scripts vary the grade of the polynomial. Here not only the grade is varied, but also variants with only containing even or odd exponents are tested. As stated above, the libraries are implemented using double precision arithmetic. Therefore, the precision of the coefficients is always selected as double. In most cases, the polynomial shall not cover the complete range of the reduced argument as especially if the function has a root at the bound, it is extremely difficult to calculate the infinity norm of the relative error close to this bound for a higher degree polynomial. With a constant or linear approximation, the infinity norm of the relative error usually can be easily calculated using analytic methods. Therefore, the complete design range is covered by a combination of a constant / linear approximation and the calculated polynomial. In order that the magnitude of these errors fit, the input range is adjusted in an iterative process.

Details on the underlining algorithms of the `fpminimax` function, as introduced above, can be found in [73]. In comparison to this approach a Taylor polynomial, as used for example in [25], has the downside that here the error grows with the distance from the origin. Algorithms like the Remez algorithm [5] compute a polynomial such that the maximum absolute error of the polynomial approximation is minimized over the complete range. But algorithms like Remez do not consider the fact that the coefficients of the polynomial are expressed as floating-points values with a limited precision. Instead normally real numbers are calculated, which then must be rounded to floating-point numbers. The algorithm, on which the `fpminimax` function is based, already considers the fact that the coefficients are stored in a certain number format, so a higher precision can be reached. In Figure 4-2 the difference of the relative error for an `fpminimax` and a Remez approximation with the coefficients are rounded to double precision is shown. The cosine approximation with even exponents up to the grade 12 is used as an example. In order to visualize the result in a better way, here the sign of the relative error is kept. In this case the performance in the worst case, for inputs close to zero, of both approaches is quite similar. But for bigger input values the performance of the `fpminimax` algorithm is clearly better. Figure 4-3 shows the result of a Taylor series for the same problem and same polynomial degree. Here the maximum error is four orders of magnitude bigger. This

is also the reason why the result is shown in a different figure. Here the error does not oscillate, instead it is monolithically increasing.

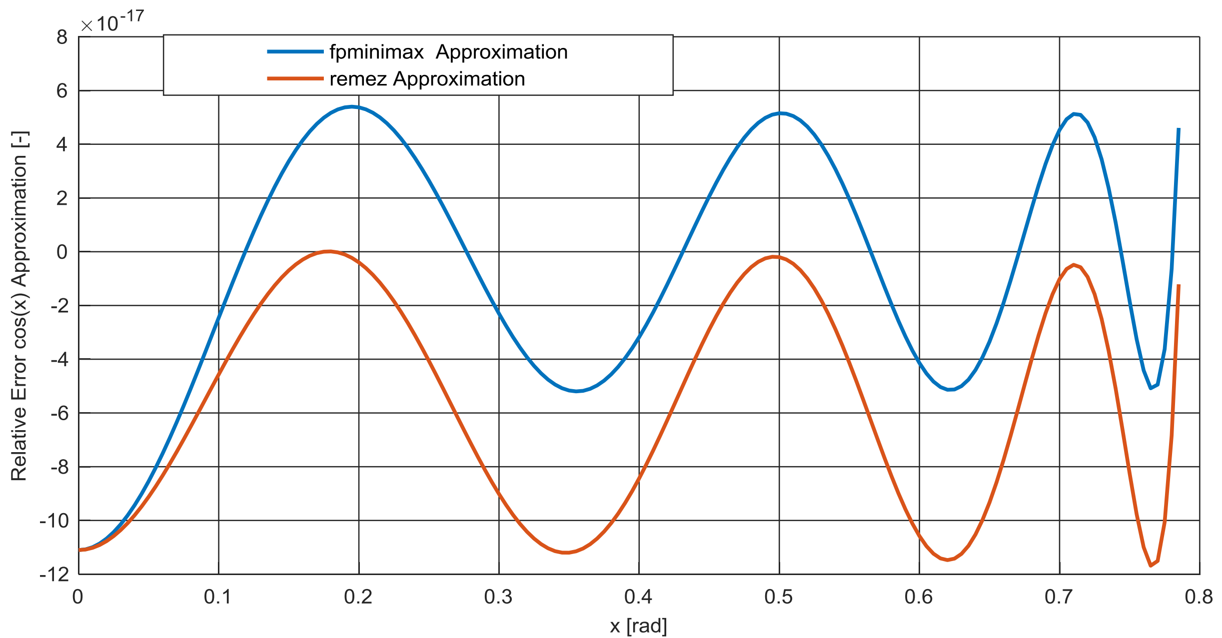

Figure 4-2: Relative Error: `fpminimax` Approximation vs. Rounded Remez Approximation

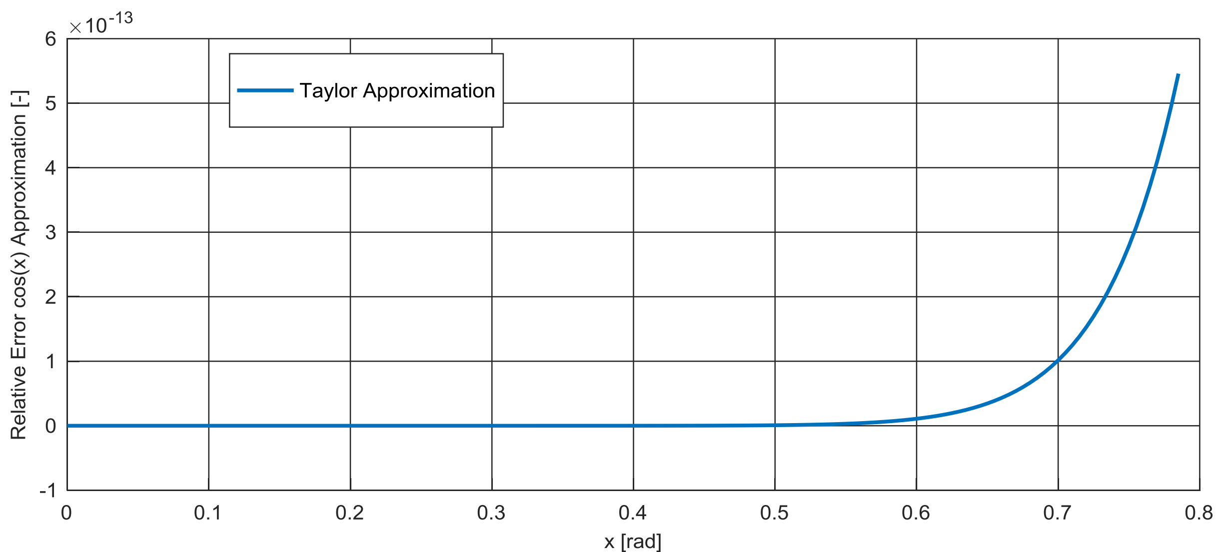

Figure 4-3: Relative Error Taylor Approximation

Next to the design of the algorithm, Sollya is also used to calculate the approximation errors of the polynomial approximation. The used function of Sollya is called supnorm. The function returns a safe bound for the error of a polynomial approximation for a given function within a certain interval. With the function `supnorm` both the relative and the absolute error can be calculated. To get an interval for the error the calculation is based on higher degree polynomial approximations of the input function. After the calculation of the error bound, the validity of this bound is checked. In case the check fails the original input interval is split to more subintervals and it is tried to solve the problem for these intervals separately. This process is repeated until the problem could be solved or the minimum interval size is reached. The minimum interval length is controlled by the parameter diam. In case the problem could also not be solved for the smallest possible interval size, an error is returned. More details on the complete algorithm

can be found in [74]. In most cases the relative error is applied, but for a more detailed analysis like the precision of the differential quotient, also the absolute error is calculated. For the cases where a special stub for the runtime error analysis is created again some additional statements might be necessary to later proof the correctness of the applied stub. Although, it is ensured that the approximation polynomials consist out of numbers which are representable as double precision numbers, the calculation of the errors in Sollya is done with higher precision. Therefore, the rounding errors that will occur on the target are not considered in the errors calculated by Sollya.

In order to incorporate the rounding errors of the evaluation on the target, an additional step is required. For this step the tool Gappa [28] is used. By using Gappa the user is supported by the generation of formal proofs. Here the same approach for the usage of Gappa as proposed by [75] is applied. Each Gappa script consists out of three sections: First, a transcript of the actual implementation. In the next section the mathematical definition of what is actually implemented follows. In the last section the given boundaries and approximation errors are defined. In addition to that the goal of the proof is defined here. The goal of the proof is normally the relative error, which can be achieved by the evaluation of the approximation using double precision calculation. In case special conditions for the stub of the function are necessary, more theorems are introduced in this section, for which a proof is generated, too. Also the naming convention introduced by [75] is used here. Variable names are mainly taken from the source code and for the mathematical exact counterpart the variable name is extended by a capital M as prefix. Normally the following names are used: x and respectively Mx for the input and y / My for the final result.

In order to make the results of the algorithm development available in the actual target, the functions are implemented in C source code. Here special care is taken to only use standard C implementation that the functions remain easily portable to different target platforms. In order to still utilize some special hardware functions these are encapsulated in in-line functions. More details on this can also be found in section 4.2. As the execution time is also considered for the selection of a specific algorithm, after the source code is developed, the code is also compiled and a WCET analysis is executed. In order to consider that the WCET of binaries executed on complex CPUs is not context independent, for this specific analysis every memory access was considered as cache hit. That this assumption is valid was shown during the integration of the elementary math function in the overall model. As shown in chapter 9 it is worth to lock all the libraries functions into the cache. Due to this locking, also the cache hit assumption for the WCET analysis during the algorithm selection is valid.

Another goal of this thesis is to enable an enhanced workflow in the context of a model-based development approach. Therefore, it shall be possible to integrate the generated libraries into the model of the application. For this integration, different methods can be used. The function can be either imported by the legacy code tool, where it is compiled and dedicated model blocks for the functions are generated. Or the code replacement library functionality can be used, which replaces the normal ANSI C library function call by the specific implementation [69]. Both methods have downsides and benefits, which are shortly described here, but not evaluated completely in this thesis.

The biggest benefit of the implementation via the legacy code tool is that it can be ensured that, beside of some target specific functions, the same code is used during the simulation as later on the target. In case the target specific functions are also replaced by implementations with bit equivalent results in the simulation, the evaluation of a test case executed in the

simulation environment can achieve a fully identical result as the execution on the target. The benefit of using the approach of a code replacement library is that in this case the nonfunctional behavior, which is stored with some Simulink blocks, is remaining in the model information. One example for this is the linearization information. As in the process at hand one model is used for the complete development from the controller design down to the target code generation, it must be ensured that the model can be linearized as this is mandatory for the analysis of the control algorithm. Some of the blocks, like for example the sine, have attached the analytical linearization. Therefore, the linearization can utilize this information so that it is faster and more precise.

In addition to the formal proof of the proposed algorithms, a verification via a test of each library is performed. The tests are executed using the processor in the loop (PIL) system presented by [76]. The PIL system enables the execution of certain test cases in parallel on the target hardware and the model simulation. The necessary exchange of data is ensured via the debugger. Therefore, no instrumentation of the target code is necessary. The test can be executed with different flavors. The first one is to compare the here proposed implementation running on the target with the original version executed during a Simulink simulation. As the Simulink implementation is lacking a detailed precision definition, this version actually does not give a result on the precision of the target algorithm. But in case the model used for the code generation is first developed using the original blocks, this test can give an estimation how close the target evaluation matches the simulation results. The second variant can compare the target version against a simulation result of a function generated with the legacy code tool. The purpose of this test is to show that the target function and the special implementation in the model actually give back the exact same result. The goal of the third variant is to demonstrate the actually reached precision. This test is not completely possible in the Simulink environment. In order to evaluate the precision reached by the implementation the results are stored and exported to an environment where the precision of the applicable mathematical function is known. In the proposed process this environment is Sollya. In order to enable an easy evaluation of the tests, a MATLAB script is prepared which reads the test results and automatically generates a Sollya script to calculate the errors. In Figure 4-4 an exemplary PIL result of the proposed double variant for the cosine is shown. The result of the target evaluation is compared to the correctly rounded values of Sollya. The graph already demonstrates that especially for integer multiple of π it is difficult to calculate a correctly rounded result. In case the PIL compares the target evaluation with the legacy code function an exact match of the results can be shown.

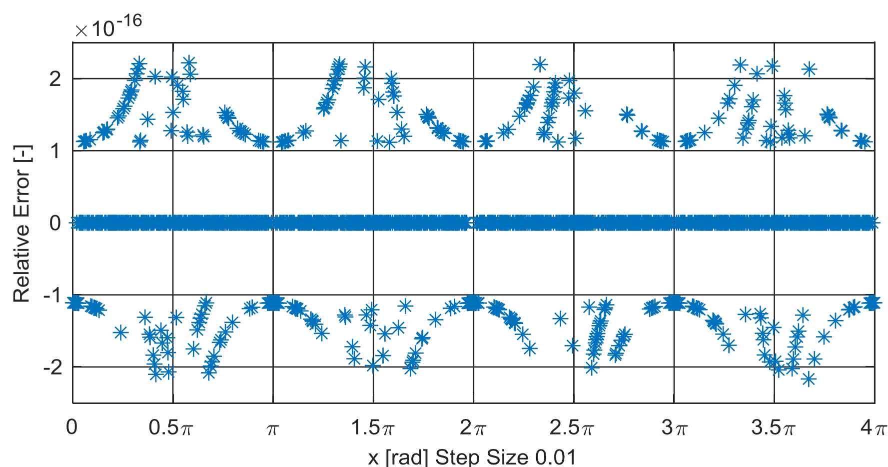

Figure 4-4: Exemplary PIL Result

The stubs for the runtime error analysis are implemented using the Astée feature to replace certain functions calls by a call to a different function. The stubs are implemented in a dedicated file, which is only available in the workspace of the runtime error analysis. This approach enables the use of unchanged original source code. The stubs are implemented using C statements and some special extensions custom to the runtime error analysis tool. An example for such extension is the expression __ASTREE_known_fact which can be used to express certain relations between variables of statements [77].

4.2 Architecture and Implementation Considerations

As stated in section 1.4, the developed library shall be easily portable between different target platforms. This is ensured by the use of pure ANSI C [47] code in the function body. But as the library shall be fast and accurate, it makes sense that specific hardware functions are utilized. Especially for frequently called functions like the FMA or calculating the absolute value, using the hardware might bring a huge benefit. In order to ensure the use of such hardware dependent options two different options exist:

- Directly use the corresponding assembler instructions
- Use of special language extensions if these are provided by the compiler

Both methods might require a specific implementation on a certain hardware and also might be dependent on the used compiler. In order to ensure an easy adaptation, these functions are grouped together and implemented in a special hardware abstraction layer. This abstraction layer basically consists out of three different categories of functions. The first one are functions specified in the ANSI C [47], which are called from the application level. The second category are also ANSI C functions, but they are not called from the application at hand, and are only used inside of the libraries. To the third category those functions belong, which are provided by the hardware and where no ANSI C equivalent exists. In order to ease the identification of functions belonging to the hardware abstraction layer all functions of the second and third category use the prefix “cpu_” in the function name. In order to omit an impact on the application software for functions of the first category the function prototype specified in the ANSI C is kept. Figure 4-5 gives an overview of this architecture:

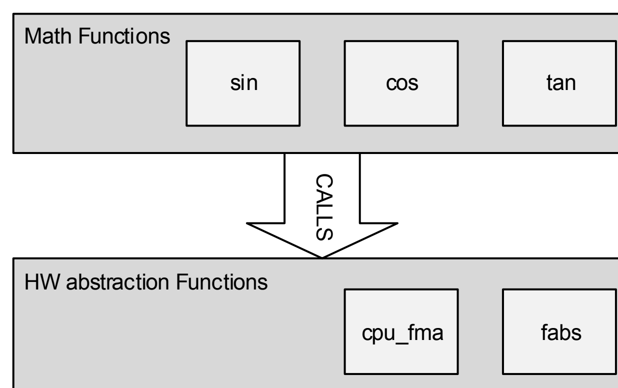

Figure 4-5: Library Architecture Overview

To avoid timing penalties, the functions are implemented as in-line functions. During compilation, the body of the in-lined functions is directly included in the caller. Therefore, no prolog and epilog is generated for the function at the executable object code level. So, it does

not make a difference if the function is in-lined or the applicable commands are directly included in the parent function at source code level.

The approach of a hardware abstraction layer is also beneficial for the integration of the library functionalities into the model-based environment, as for simulations the model is normally executed on a different target.

An example for a function contained in this abstraction layer is the FMA instruction. As already mentioned in section 2.2 this function calculates $(x * y) + z$ without intermediate rounding. Due to the less rounding operations, it brings a benefit in precision but also for the execution time on some platforms. As also contained in the IEEE 754 [3] the FMA instruction is quite common to many targets [15, 42, 43]. In the ANSI C standard no direct operator is defined for this function [47]. In order to ensure that this operation is actually called, and not implemented via a separate multiply and add instruction, a special hardware abstraction function is introduced. As the simulations normally are executed on Intel processors, the abstraction of the fused multiply add is of special interest. As Intel processors provided this functionality natively since the year 2013 [43], the hardware can now be utilized during simulation in case a special hardware abstraction layer is provided for the CPU. With the assumption that the hardware and the compiler do not introduce bugs, the hardware abstraction level should enable an efficient simulation on the host and execution on the target with equality to the last bit. Especially in an environment where simulation is an important part to evaluate the design this is a huge benefit.

For the implementation of the elementary functions also some architectural decisions must be taken. Many elementary functions consist out of the following three sub steps:

- Argument reduction
- Approximation on the reduced range
- Reconstruction of the result on the reduced range to the original range

In case a complete modular approach is chosen, these three steps might be integrated into different sub-functions. In the implementation a different approach is chosen. For most functions, all three different steps are included in a single C function. There are various reasons for choosing this approach. First, the functionalities implemented in the sub-functions are quite special. Therefore, a reuse of these functions will not occur at all or might be quite limited. For example, the range reduction is in principle equal for the low and high precision variant of a function. But for the low precision variant often a lower precision for the range reduction is sufficient. For that reason, even in these cases there will not be a hundred percent reuse. The benefit of implementing the complete functionality in a single function is that first the prolog and epilog of different sub-functions are not necessary. Second, a better optimization by the compiler can be achieved. Compilers normally apply the optimizations at source code function level. In the case that the complete implementation is contained in a single function, the optimization might lead to better results. For example, the utilization of the available registers then might be better. This not only applies to the compiler, even on source code level some functions can be merged. For example, the last multiplication of the approximation can be joined with an addition in the reconstruction step in a single FMA instruction. In case the two steps are included in two different C functions such a merge might be quite difficult to understand. But in case all is contained in the same function, it is much clearer. The only exception to this direct implementation concept is the approximation on the reduced range of

the sine and cosine. The reason for the deviation in this case is that both sine and cosine are based on the same approximation, so here indeed the benefit of reusing comes into account.

Another aspect supporting the portability is the use of floating-point numbers. In contrast to other libraries the implementation here tries to use floating-point operands whenever possible. In order to omit problems like cancelations or the utilization of higher precisions in other projects floating-point operations are implemented on bit level [10]. As the IEEE 754 [3] does not specify any memory layout for storing the numbers the bit representation might change between a little and big endian system. In case operations are implemented on bit level, these potentially also must be adapted. This is omitted here by mainly using floating-point operators and using robust algorithms to avoid problems, which might be introduced by floating-point operations.

In most implementations of the elementary functions, only the positive inputs are considered for the approximation and the sign of the result is added in a final reconstruction step using properties like if the approximated function is odd or not. For example, for the approximation of the arcus tangent such an approach is used.

Listing 4-1 shows the C source code used in such cases. The absolute value of the input is generated and in addition a Boolean flag, indicating the sign of the original input.

```
1   if(x < 0.0)
2   {
3     x_abs = -x;
4     sign = CPU_TRUE;
5   }
6   else
7   {
8     x_abs = x;
9     sign = CPU_FALSE;
10  }
```

Listing 4-1: Code Pattern to Extract Sign and get Absolute Value

An alternative approach to get the absolute value could be to remove the assignment to “`x_abs`“ in the “if” and “else” branch, and use instead “`x_abs = fabs(x);` “. But in cases where `x` is not further used in the function, the register allocation of the compiler did not give the best result for the alternative solution. In the implemented version “`x_abs`“ is calculated as follows: `FA` and `FB` representing two different floating-point registers. In the if path the compiler produced an “`FA = fneg(FA)`“ and in the else path nothing is done. So only one floating-point register is used. In case “`fabs`“ is used, the input output registers are different: “`FB = fabs(FA)` “. Therefore, the function might use an additional floating-point register. In case for this register a non-volatile register is used. The original value must be stored on the stack and restored in the function epilog. This will lead to additional instructions, which are accessing the memory and so increase the WCET.

In order to achieve efficient implementations with a low WCET also divisions are omitted. The reason therefore is that the floating-point division has a much higher execution time. Most of the floating-point instructions have a latency of three or four clock cycles. The single division has a latency of 18 and the double division goes up to 33 cycles [15]. Next to the fact that the latency of the instruction is much higher, it is even worse as they block the FPU pipeline. This leads to a much lower throughput in case of consecutive floating-point instructions. For example, the floating-point add instruction has a latency of three cycles. But three independent consecutive add instructions need due to pipelining in the optimal case only five and not nine

cycles. The execution of three double precision floating-point divisions indeed takes 99 cycles as this instruction is not pipelined. Considering these aspects, it is very beneficial if divisions can be omitted. For instance, this is done by directly multiplying with the inverse during the range reductions. Or for the calculation of the square root the complete algorithm is chosen such that multiple divisions are omitted.

4.3 Relation to Certification Aspects

Due to the special conditions under which the research aircraft is operated, there is no need to actually qualify the software at hand, but the proposed approach shall be transferable to projects where an actual certification is mandatory. In order to enable this, the certification considerations need to be taken into account. Therefore, in this section the certification aspects are addressed. The guidance for certification of airborne software in general is given in DO-178 [18]. Next to this document there exist special supplements: One is taking care of the model-based aspects [78] and another one of formal methods [79]. Although, it is proposed to integrate the developed elementary functions in the model-based development approach, the proposed process does not impact special objects of the model-based certification workflow. Considerations regarding the model-based certification workflow can be found in [67, 80]. Here the workflow of the elementary function development is considered. As described in the previous section, some tools are used to develop and validate the algorithms of the elementary functions. For the usage of tools the DO-178C refers to the DO-330 [81]. The DO-330 gives guidance for the qualification of tools. To incorporate that failures in tools might have different severity, five tool qualification levels (TQL) are defined. Ranging from one, which expresses the highest qualification effort, to five, with the lowest qualification effort. Which level is applicable for a certain tool is actually not defined in the DO-330, as this guidance is directly given in the DO-178. The allocation to a certain TQL is based on the DAL of the airborne software and on three criteria, which consider the impact of errors of the tool.

Criteria one tools are applications where the output becomes directly part of the airborne software and which can introduce a wrong functionality of the produced software. An example for such a tool is a code generator where the output is not further verified. To criteria two tools software is mapped which automates certain verification process and where the output is used to reduce other development processes or verification processes. An example for a criteria two tool is: A utility verifies the compliance of the source code to software architecture and low level requirements and as this check handles the complete interaction between these development artifacts the check for traceability between this two artifacts is omitted. Criteria three tools are classical verification tools, which potentially could fail to detect an error in the domain of their expected use. An automated check if a certain code coverage is already reached is an example for such a tool.

To allocate the TQL the DO-178 contains a matrix. The rows of the matrix are the different DALs and the columns correspond to one of the three impact criteria. Where the combination of DAL A and criteria one is the most rigorous and DAL D with criteria three is the most relaxed combination.

Now the question arises: Are the tools, which are used in the proposed workflow, subjected to qualification? In order to answer this, special care needs to be taken what is the actual outcome of each tool and where is this outcome actually used.

The first tool in the proposed workflow is Sollya [72]. This tool actually gives a good approximation polynomial of a function and an infinity norm of the approximation error. As the proposed polynomial actually becomes part of the final software, some might think that the function generating this polynomial might be subjected to qualification. Actually, this is not the case. Although it is desired that the used Sollya function for generating the polynomial returns, with the given boundary conditions, a good approximation. For the certification it is not relevant if it is the best or the worst possible approximation. The only thing which is of interest, is how good the proposed approximation actually is. In the case of Sollya, the actual precision of the approximation is calculated using a different function. As this information is further used in the workflow, the correctness of this function indeed needs to be shown. This can either be achieved by manually verifying the given result or by a qualification of this specific function. Qualifying only this specific function will be a much lesser effort than qualifying also the generation algorithm, which is much more complex as it returns an optimized result. As this result is only used for V&V activities and the result is not further used to replace other activities, it is expected that a TQL of five would apply to this functionality.

The next tool in the chain is Gappa [28], which is used to generate the proofs showing the total reached precision of each implementation. As none of the outputs of this tool become actually part of the target software a criteria one tool can be excluded at the beginning. But as the tool supports the validation, still some lower category might be applicable. In order to determine this a detailed look on the outputs of the tool is necessary. The tool indeed helps to generate automated formal proofs in order to show that the desired requirements are met. In case these formal proofs are taken, without further verification the tool indeed would be subjected to qualifications. But this is not how the designers of the Gappa tool expected their tool to be used. Instead as taking the generated proof as correct, the proof can be exported in various formats and then checked by a second instance. The way leading to the lowest tool qualification, would be to export the proof in the Latex format and then transform it to a human readable format, which is then checked manually. In case many proofs are compiled and need to be checked this might not be the most efficient solution. Therefore, also an export to the Coq proof engine [82] is possible. Coq is a management system for machine-checked formal proofs. When the generated proofs are verified by Coq then the tool qualification would be necessary for this tool. As the tool does automate a verification step also a TQL of five is expected for this tool. Also the formal verified compiler CompCert [71] is based on Coq. Therefore, in conjunction with showing the correctness of the formally verified compilation there might be some synergies in a potential tool qualification.

In the overall development cycle, also the tools aiT [53] and Astrée [77] are intensively used. The application aiT is used for the WCET analysis and with the help of Astrée the runtime error analysis is executed. The novelty regarding these tools in this thesis is the interactive use during the development, for example like the specific stub generation for the elementary math functions. This has not an impact on the actual tool itself. Both tools are quite commonly used in avionics projects [83–86]. In order to support this, the tool vendor also provides a qualification kit [53, 77], which can be applied in this case.

5 Trigonometric Functions

Trigonometric functions are essential in flight control software as they for example play an important role in the calculation of positions or the transformation between different coordinate systems. This also explains the high number of call sites presented in the table in section 3.5, but also the amount of calls on the WCET path is high. The WCET contribution of all trigonometric functions is $629\ \mu s$, which corresponds to a contribution of $11\ \%$ to the WCET of the complete flight control software without any cache locking. This clearly shows the necessity that an efficient implementation of these functions is necessary to come to an overall good performance. As the methods applied to approximate the trigonometric functions are all quite similar, the implementation of all of these are shown in this chapter. All trigonometric functions which are actually used in the software and listed in Table 3-1 are introduced in this chapter.

5.1 Sine and Cosine

As shown in Table 3-1 the sine and cosine are the most frequently called trigonometric functions. These functions can be defined as the ratio of the opposite leg and the hypotenuse in the rectangular triangle for the sine and the ration of adjacent leg and the hypotenuse for the cosine. An alternative analytical definition is [11]:

$$\sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \tag{5-1}$$

$$\cos(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!} x^{2k} \tag{5-2}$$

The sine and cosine are periodical functions. Due to the phase shift of $\pi/2$ both implementations can be based on the same approximations.

5.1.1 Input Restrictions

As specified in the ANSI C [47] the implementations here shall return the sine or cosine of an input argument given in radians. The verified precision shall be achieved for input arguments in the range of $[-4\pi, 4\pi]$. This range is selected as angles are normally given in the range of $\pm\pi$ or $[0,2\pi]$. For cases where the direction of the angle is important, also the range of $[-2\pi, 2\pi]$ might be used. With the selected input range it is possible to also handle cases where the input is directly the sum of two angles. As the sine and cosine are defined for the complete range of normalized and de-normalized double numbers, the definition of the behavior in special cases is not applicable. In order to get a robust behavior for out of range inputs, the precision of the range reduction should not decrease dramatically for out of range arguments. For variants which reach a precision up to $e_{tot}^{rel} = 2^{-24} = 5.96 * 10^{-8}$ the input shall be a single precision number, for higher precisions a double precision number shall be chosen.

5.1.2 Architecture

In order to discuss the potential architecture of the sine and cosine it is beneficial to visualize the functions in the unit circle as shown in Figure 5-1.

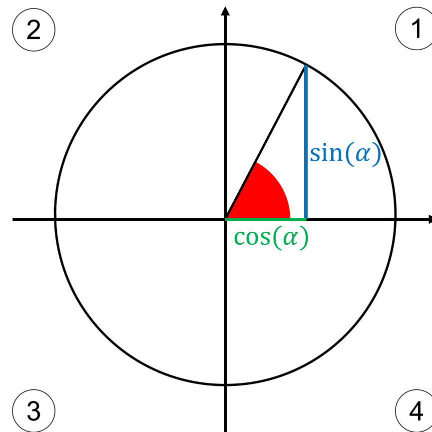

Figure 5-1: Sine and Cosine in the Unit Circle

As the sine and cosine functions are periodic functions it is sufficient if one period is approximated. But as the functions also have some symmetries, this range can be further reduced. Two possible variants of the range reduction are to reduce the input to the range of $\pm\pi/2$ or $\pm\pi/4$. These ranges are smaller than the actual period of the functions, which is 2π. In order to reach to these ranges an additive range reduction is executed. In addition to the reduced input argument for the approximation, some conditions are needed to reconstruct the input argument. In the following these are presented.

In the context of the unit circle, which is shown in Figure 5-1, it is obvious that the values for sine and cosine will repeat with the same or a different sign for all the quadrants. So, it is intuitive to do the approximation in the range of a single quadrant and reconstruct the sign dependent on the applicable quadrant. It is important that the mapping is done without any sum or difference in order to not extinguish the precision reached during the range reduction. In Table 5-1 an overview is given for the cosine function how the factor n of the range reduction and the sign of the reduced range can be used to map the result to a quadrant.

Quadrant	4	1	2	3
$n = \left\lfloor \frac{x}{\tilde{\pi}} \right\rceil$	$2m \; with \; m \in \mathbb{N}$	$2m \; with \; m \in \mathbb{N}$	$2m + 1 \; with \; m \in \mathbb{N}$	$2m + 1 \; with \; m \in \mathbb{N}$
$x_{red} = x - \left\lfloor \frac{x}{\tilde{\pi}} \right\rceil \pi$	< 0	≥ 0	≤ 0	< 0
$\cos(x)$	$\cos(\lvert x_{red} \rvert)$		$-\cos(\lvert x_{red} \rvert)$	

Table 5-1: Mapping from the Range Reduction to the Cosine Approximation for $[0, \pi/2]$

As shown in the table above the combination of the sign of reduced argument and if the factor n is odd or even is unique for each quadrant. But as the cosine is an even function, actually it is sufficient to only consider the absolute value of reduced argument and reconstruct the result of the cosine based on the fact if the factor n is odd or even. The even or odd check of an integer number is quite easy to implement in C. It can be implemented in a simple check if the result of a “bitwise and” with one is zero or not.

The second variant of the range reduction reduces the input argument even further and divides the quadrant into half, which results in a reduced range of $\pm\pi/4$. In case the phase shift of $\pi/2$ between the sine and the cosine is considered, the whole input range can be approximated, with a sine and cosine approximation on the interval $[0, \pi/4]$. In order to implement a mapping from the reduced argument to the appropriate polynomial approximation Table 5-2 gives an

overview of the mapping for the first eight intervals. These first eight intervals represent a range of 2π, which is equal to the period of the functions, so the mapping repeats afterwards.

Range of input argument x	$\left[\frac{7\pi}{4}, 2\pi\right[$	$\left[0, \frac{\pi}{4}\right[$	$\left[\frac{\pi}{4}, \frac{\pi}{2}\right[$	$\left[\frac{\pi}{2}, \frac{3\pi}{4}\right[$	$\left[\frac{3\pi}{4}, \pi\right[$	$\left[\pi, \frac{5\pi}{4}\right[$	$\left[\frac{5\pi}{4}, \frac{3\pi}{2}\right[$	$\left[\frac{3\pi}{2}, \frac{7\pi}{4}\right[$
$n = \left\lfloor \frac{x}{\frac{\pi}{2}} \right\rceil$	4	0	1	1	2	2	3	3
$x_{red} = x - n * \frac{\pi}{2}$	< 0	≥ 0	< 0	≥ 0	< 0	≥ 0	< 0	≥ 0
$\cos(x)$	$\cos(-x_{red})$	$\cos(x_{red})$	$\sin(\lvert x_{red}\rvert)$	$-\sin(x_{red})$	$-\cos(-x_{red})$	$-\cos(x_{red})$	$-\sin(-x_{red})$	$\sin(x_{red})$
$n\%4$	0		1		2		3	
$\cos(x)$	$\cos(x_{red})$		$\sin(-x_{red})$		$-\cos(x_{red})$		$\sin(x_{red})$	

Table 5-2: Mapping from the Range Reduction to the Cosine Approximation for $[0, \pi/4]$

Similar to the case before, sums and differences must be omitted in the mapping. The table shows, similar to the previous case, that for each value of the factor n and the sign of the reduced argument a unique allocation to a sine or cosine function, based on the reduced argument, is possible. If at the same time the properties are used that the sine is an odd function and cosine is an even function, the mapping can be further simplified. In this case modulo four of the factor n can be allocated to an approximation algorithm on the reduced range, as shown in the last line of Table 5-2.

Table 5-1 and Table 5-2 only show the mapping for the cosine. The mapping for the sine works similar. In the following paragraphs mostly the cosine function is discussed but the considerations are directly transferable to the sine function.

5.1.3 Range Reduction

For the sine and cosine an additive range reduction is necessary, which can get quite complex for large input arguments [5]. But as the sine and cosine in this context are designed for an embedded system with well-known input ranges, the input to the range reduction can be limited to $[-4\pi, 4\pi]$. The relatively small range enables the possibility of simple range reduction algorithms, which is essential to get a computation-power efficient algorithm. As stated above, the precision proof is necessary for the range of $[-4\pi, 4\pi]$. But further on, it is desired that for values out of this range a quite good estimate is achieved. As described above, two different variants of the core algorithms are evaluated one approximating the range $[0, \pi/2]$ and another one approximating the range $[0, \pi/4]$. If the sign of the reduced input argument is considered in the mapping to the core algorithm, a range reduction to the range $[-\pi/2, \pi/2]$ or $[-\pi/4, \pi/4]$ is necessary.

As first step of the range reduction the absolute value of the input value is taken. As the cosine is an even function, here only the absolute value using the `fabs` function is calculated. For the sine the sign of the input argument is necessary for the result reconstruction. Therefore, the code pattern presented in section 4.2 is used to get the absolute value and an indication if the input is a positive or negative value.

Depending on the format of the input argument, two different range reduction methods are applied. The first one is used in the case of single inputs which correlates to a lower precision of the overall algorithm. In this case, a high precision range reduction does not bring any benefit. Therefore, only double precision constants are used to reduce the range. In case a

higher precision should be reached, a range reduction using only double precision constants does not provide enough precision. For that reason, in this case the approach of Cody and Waite [5] is used, where the constant of the range reduction is implemented by two floating-point numbers. When the Cody and Waite algorithm is implemented using FMA instructions, the range can be further extended [87].

In the following only the range reductions to the interval $[-\pi/4, \pi/4]$ are presented. The ones to the interval $[-\pi/2, \pi/2]$ work similar. Especially as $\pi/4$ and $\pi/2$ differ only in the exponent and have the same significand in the binary based floating-point representation.

As first step for both range reductions the integer factor n must be determined such that $(x - n * K) \in [-K/2, K/2]$. This is done by multiplying x by double precision representation of the inverse of K. In order to get the rounding to the nearest integer, 0.5 is added to the result of the multiplication and then a cast to integer follows.

Afterwards, the first reduction step is performed by calculating $x_{red1} = x \ominus n \otimes C_1$ with C_1 being the double precision representation of K. In case of $K = \pi/2$ the round to nearest double precision number has three consecutive zeros in the LSBs of the significand. This leads to the fact that if the calculation is executed using double precision arithmetic, the product can be exactly represented and calculated without rounding error for $n \leq 2^3 = 8$. With Sterbenz lemma, as introduced in section 2.2, it can be shown that the subtraction is also exact. Therefore, for $|x| \leq 4\pi$, which is the design range, the error made by the first step of the range reduction is only due to the fact that C_1 is not exact K. In order to get a robust behavior also if x is out of the design range, the calculation of x_{red1} can be implemented by using a FMA instruction.

```
1  /*Calculate integer multiple of pi/2 rounded to the nearest of input*/
2  factor = (CPU_INT32S)(cpu_fma(K_inverse,x_abs,0.5));
3  /* Use Codey and Waite Range reduction*/
4  x_red = cpu_fma(factor,minusC_1,x_abs);
```

Listing 5-1: First Step Range Reduction

The intermediate result of the multiplication is not rounded to 53 bit in this case, but represented using 106 bits. So, the following theorem of [87] can be used:

“Let a and b be two floating-point numbers with $p + l$ bits in the significand, where $l \geq 0$.If

$$\frac{b}{2} \leq \left(1 + 2^{-l}\right)^{-1} b \leq a \leq \left(1 + 2^{-l}\right) b \leq 2b \tag{5-3}$$

then $a - b$ can be represented exactly by a floating-point number with p bits in the significand.”

Considering also the other constraints in the paper [87], it can be shown that using a FMA instruction $x_{red1} = x - n * C_1$can be calculated without any rounding error up to $|n| \leq 1.5 * 10^{17}$. As $1.5 * 10^{17}$ is bigger than the maximum value of a 32 bit integer number, actually this limit applies here instead.

Considering the fact that no rounding error in the calculation of the first step of the range reduction occur, the absolute error of the first reduction step can be calculated as follows:

$$\left|AbsErr_{RangeRed}\right| = |n * (K - C_1)| \tag{5-4}$$

$$for\ K = \frac{\pi}{2}\ and\ C_1 = \widetilde{K}: K - C_1 \le \frac{1}{2} ulp(K) = 2^{-53} \tag{5-5}$$

$$For\ |n| < 8: \left|AbsErr_{RangeRed}\right| \le 2^3 * 2^{-53} = 2^{-50} \tag{5-6}$$

As a FMA instruction for the range reduction is used, the fact that no rounding error occurs during the calculation is valid for a bigger input range. Therefore, the result of equation (5-6) can be extended to bigger input ranges. As the initial rounding error is multiplied by the factor of how often $\pi/2$ fits into the input range, for bigger input ranges the absolute error grows linear. As this is a continuous behavior the function can be considered as robust.

As shown in (5-6), the absolute error of the first range reduction step for input arguments within the design range is 2^{-50}. It needs to be determined if this error is already sufficient to get a result with the expected precision. For high precision input it is easy to show that this is not the case. As C_1 equals $\widetilde{K}$, the double representative of K and not K, catastrophic cancelation will occur for values close to $\pi/2$. The catastrophic cancelation might lead to a result with no or very view correct bits in the significand. In case only the first range reduction step is applied $\cos(\widetilde{\pi/2})$ would be zero and not $6.1e-17$. This leads to a relative error of one, which is much too high. In the case of the lower precision algorithms, it is more difficult to evaluate if the first step is already sufficient. As the lower precision algorithms have a single precision number as input, catastrophic cancelation might not occur as the internal calculations have a higher precision. As the overall goal is to achieve a low relative error, here the area close to integer multiples of $\pi/2$ is of interest. Only the case where the function return value is close to zero must be considered, as here the absolute error has much higher impact on the relative error for small values. Figure 5-2 shows for instance the detail next to an x value of $\pi/2$ for the cosine function. The situation is an example for all roots of the sine and cosine functions. The cases where the function has a positive gradient are similar. The figure exemplarily shows the actual cosine function, with a curvature and a root at $x = \pi/2$. Close to the root the sine and cosine are approximated by a linear function with a gradient of plus/minus one (Compare: 5.1.4). This approximation is shown for the real number case with a root at $\pi/2$ and the floating-point number case with a root at the double precision representative of $\pi/2$. Next to these curves the single representative of $\pi/2$ is printed on the x axis.

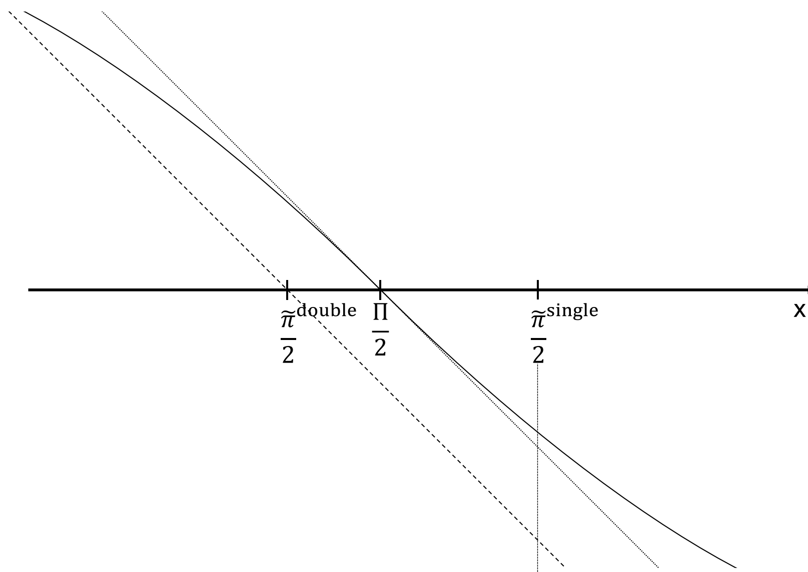

Figure 5-2: Cosine close to $\pi/2$

As shown above, during the calculation of multiples of $\widetilde{\pi/2}$ no additional rounding error occurs. So, the following equation is valid for the complete design range:

$$\widetilde{n * \frac{\pi}{2}} < n * \frac{\pi}{2} \; for \; integer \; n \; in \; [1,8] \tag{5-7}$$

For the single numbers this is actually not true. In this case the number closest to an integer multiple of $\pi/2$ can be either bigger or smaller than the real number. But the number will never be equal to the double representative. According to the figure above it is obvious that the cases where the single representative is bigger than the real number and on the opposite side than the double representative are the worst case for the error. The following components are necessary for the calculation of the overall relative error:

- The actual function value for the single representative
- The value of the approximation function for the single representative

With r equal to $n * \pi/2 \; for \; integer \; n \; in \; [1,8]$ the value can be approximated by:

$$f\left(\tilde{r}^{single}\right) > \left(r - \tilde{r}^{single}\right) * \left(1 - e_{apr}^{rel}\right) \tag{5-8}$$

With e_{apr}^{rel} representing the relative approximation error with respect to the function f. Close to integer multiples of $\pi/2$ the function is approximated by a linear function with the slope one, so the value of the approximation can be calculated as:

$$f\left(\tilde{r}^{single}\right)_{apr} = \tilde{r}^{double} - \tilde{r}^{single} \tag{5-9}$$

Incorporating the result of precision considerations of the range reduction, the equation resolves to:

$$f\left(\tilde{r}^{single}\right)_{apr} = r - 2^{-50} - \tilde{r}^{single} \tag{5-10}$$

Using equation (5-8) and (5-10), the total relative error can be expressed as:

$$\begin{aligned} e_{tot}^{rel} &\leq \frac{\left|\left(r - 2^{-50} - \tilde{r}^{single}\right) - \left(r - \tilde{r}^{single}\right) * \left(1 - e_{apr}^{rel}\right)\right|}{\left|\left(r - \tilde{r}^{single}\right) * \left(1 - e_{apr}^{rel}\right)\right|} \\ &= \frac{\left|2^{-50} - \left(r - \tilde{r}^{single}\right) * e_{apr}^{rel}\right|}{\left|\left(r - \tilde{r}^{single}\right) * \left(1 - e_{apr}^{rel}\right)\right|} \end{aligned} \tag{5-11}$$

To estimate this, the closest distance between the real number and the single representative is needed. For the given input range the minimum of $\left(r - \tilde{r}^{single}\right)$ is calculated using Sollya. It resolves to $-2^{-21.72} < -2^{-22}$. Taking this value and the desired relative error into account equation (5-11) resolves to:

$$e_{tot}^{rel} \leq \frac{2^{-50} + 2^{-22} * 2^{-24}}{2^{-22}} \approx 2^{-23} = 6.3 * 10^{-8} \tag{5-12}$$

The result of equation (5-12) still is quite close to the desired result of 2^{-24}. Therefore, for the low precision variants no further range reduction step follows.

As shown above in the case of double precision input signals the precision of the first reduction is not sufficient, so a second reduction step follows. The precision of the reduced argument is

increased by the calculation of $x_{red2} = x_{red1} \ominus n \otimes C_2$ where C_2 is the double precision representation of $K - C_1$. In order to omit the intermediate rounding, this step is also implemented using a FMA instruction. To show a satisfying precision of the higher degree core algorithm, a relative error of the range reduction is necessary. The error can be calculated as follows:

$$|e_{red}^{rel}| = \left| \frac{n * \left(K - (C_1 + C_2)\right)}{x - n * K} \right| \tag{5-13}$$

With $K - (C_1 + C_2)$ being equal to the absolute error made by the approximation of K by two double precision numbers. In the case of $K = \frac{\pi}{2}$ the absolute of this error is approximately $1.5 * 10^{-33} < 2^{-109}$. The term $x - n * K$ is the minimum distance of a double precision floating-point number to an integer multiple of $\frac{\pi}{2}$. This value is determined using a Sollya script:

$$For\ |n| \in [1,8]: |x - n * K| \approx 6.12 * 10^{-17} > \ 2^{-54} \tag{5-14}$$

So, the relative error can be bound to:

$$|e_{RangeRed}^{rel}| \leq \frac{2^3 * 2^{-109}}{2^{-54}} = 2^{-52} \tag{5-15}$$

In order to incorporate the error in the final rounding step to double precision, for the proofs a precision of 2^{-51} is used for the range reduction. This value is small enough to show satisfying precision from the core algorithms over the complete design range. In case the arguments out of the design range are passed to the range reduction, the denominator of the relative error will grow linear. In addition, the numerator might decrease for bigger input arguments. With Sollya it can be shown that the numerator for values of $n < 29$ the numerator will not be smaller than in equation (5-14). For $n = 29$ the distance is approximately $2^{-60.5}$. This is already quite close to the lowest possible distance for double precision inputs. As according to [5] the double precision floating-point number closest to an integer multiple of π is $6381956970095103 \times 2^{797}$, for which the distance resolves to $|x - n * K| \ \approx 2^{-60.9} > 2^{-61}$.

5.1.4 Core Algorithm

On the reduced range, the sine and cosine function are approximated by polynomial functions. As already discussed, the two different versions of the range reduction imply two different approximation intervals. In the following paragraphs the actual approximations are examined. First, the one where the sine or cosine are approximated on the interval $[0, \pi/4]$. Afterwards, the approximation of the cosine on the interval of $[0, \pi/2]$ follows. After both polynomial approximations are deduced, a comparison based on the achieved WCET and precision follows in the next subsection.

The workflow for getting a polynomial approximation is as described in chapter 4. In the following paragraphs the specialties for sine and cosine approximation are described. For all variants polynomials of different degree are created and compared.

Cosine approximation for the interval $[0, \pi/4]$.

For values close to zero it is difficult to show by numeric methods that a certain relative error is kept. Therefore, the approximation of the cosine is split into three sections, first for the value

of zero the exact result of 1.0 is returned, then an approximation by a constant follows before the actual polynomial approximation starts. These approximations shall have a relative error in the same range. As shown by [24], for a given error e^{abs} the upper bound of the interval where the error of the approximation $1 - e^{abs}$ does not get bigger than e^{abs}, can be calculated as follows:

$$x_{max} = 2 * \sqrt{e^{abs}} \tag{5-16}$$

In difference to [24] where a certain relative error shall be achieved, for this work the approximation error is not known a priori. As polynomial of different degrees are analyzed, the achieved approximation error varies. Equation (5-16) shows that the validity of the constant approximation increases when the corresponding allowable error increases. As polynomials with a lower degree have a bigger error than the one with higher degrees a constant error bound makes no sense in this case. To get a good approximation for a given degree, the error of the constant approximation shall be equal to the error of the polynomial approximation. Equation (5-16) gives a relation to the absolute error, but the goal is to achieve a relative error of the same magnitude. As the constant approximation is only valid close to zero and there the cosine returns values close to one the simplification that relative and absolute error are treated as interchangeable here is acceptable. Also for a constant degree the error bound cannot be determined, as the error of the polynomial approximation depends on the input range and the low bound of polynomial approximation depends on the error of the approximation. To solve this problem an iterative approach was selected. The approach is implemented in a Sollya script. Figure 5-3 gives an overview of the program flow of the script used to gather a polynomial approximation.

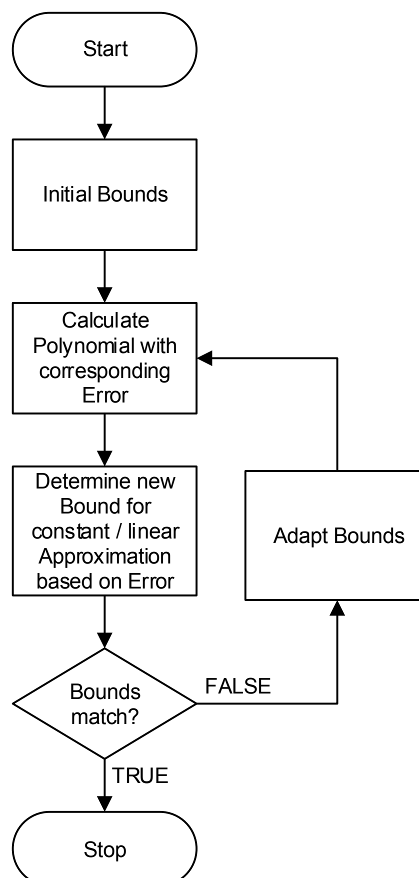

Figure 5-3: Flow Diagram of the Cosine Sollya Script

The algorithm starts with the input range $[2^{-26}, \pi/4]$. The low bound of 2^{-26} ensures that the error of the constant approximation is not bigger than one ulp for a double precision calculation. First, a polynomial approximation is calculated with the initial input interval. The absolute error is then used to determine a new low bound according to equation (5-17):

$$NewLowBound = 2 * \sqrt{\inf\left(e_{apr}^{abs}\right)} \tag{5-17}$$

As Sollya returns a small range for the error, the lower bound is used for the calculation. This is expressed by the “inf” in equation (5-17). The lower bound leads to a conservative result. The result of the calculation is rounded to a double precision number, using the round to nearest mode. In case the new low bound is bigger than the initial value of 2^{-26}, the low bound of the polynomial approximation is adapted and a new polynomial is determined. This process is repeated till the low bound is not further improved. For the following iterations the previous bound is compared and not the initial bound.

As the approximation is not based on Taylor series, the exponents can be freely selectable. Here two variants for the exponents of the polynomial are analyzed: The first one which used all possible integer exponents in the range of $0..n$. For the different variants n is varied from two to twelve. And a second one, motivated by the analytical definition of the cosine as given in (5-2), which only considers even exponents. Here the biggest exponent is chosen in a range from two to fourteen. As the Taylor series only has even exponents, an approximation based on even exponents might lead to better results, especially in case the number of necessary calculations are considered. Approximations with a higher degree still bring a benefit during the calculation of the approximation polynomial in Sollya. But with the help of Gappa, it can be shown that in case the polynomial are evaluated using double precision numbers, no further benefit is gained. For example, the analysis with Gappa shows in case of all exponents are used for the cosine approximation, the precision is not further increased for the exponents bigger than 11. The reason is that the rounding errors made during the evaluation using double precision arithmetic, are more significant than the ones made by the approximation. Figure 5-4 shows the pure approximation error (blue line) and the total error made during the evaluation with double precision numbers (green line). For small exponents the additional error made during the evaluation of the polynomial is negligible. For exponents bigger than nine the error starts growing.

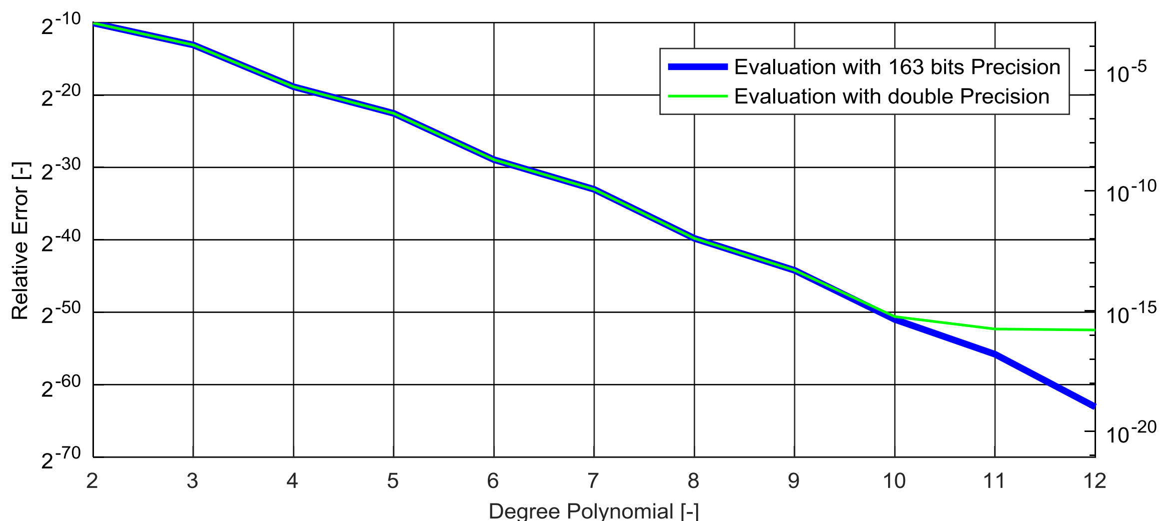

Figure 5-4: Cosine Approximation Error and Total Error

Sine approximation for the interval $[\mathbf{0}, \boldsymbol{\pi}/\mathbf{4}]$.

Similar to the cosine approximation, also for the approximation of the sine close to zero a different approximation is used than for the rest of the interval. In difference to the cosine, a linear approximation is used. As the sine of zero is zero, here the absolute and relative error cannot be treated as equal and a bound for the relative error must be given. In [24] it is shown that inequality (5-18) is valid for x in the interval $]0, \frac{\pi}{4}]$

$$x\left(1-\frac{x^2}{6}\right) < \sin(x) < x \tag{5-18}$$

Using (5-18) the relative error of the approximation $\sin(x) \approx x$ can be bounded as follows.

$$e^{rel} = \frac{x-\sin(x)}{\sin(x)} < \frac{x - x\left(1-\frac{x^2}{6}\right)}{x\left(1-\frac{x^2}{6}\right)} = \frac{\frac{x^2}{6}}{1-\frac{x^2}{6}} = \frac{x^2}{6-x^2} < \frac{x^2}{6} \tag{5-19}$$

Therefore, for a given relative error the upper bound of the validity of the linear approximation $\sin(x) \approx x$ can be calculated as:

$$x_{up} = \sqrt{6e^{rel}} \tag{5-20}$$

This leads to an initial low bound for the polynomial approximation of 2^{-26}. Similar to the cosine, the polynomials are determined with an iterative Sollya script which adapts the low bound of the interval.

For the sinus two variants of for the exponent selection were analyzed. The first, like for the cosine, using all integer exponents from zero to n with n varying from 2 to 12. Like for the cosine exponents bigger than eleven do not bring a benefit in the precision of the function if the polynomial is evaluated using double precision. The second variant uses only odd integer exponents. In order to really get an odd function, the exponent of zero is omitted. Here polynomials with the highest exponent ranging from three to fifteen were evaluated. Considering the evaluation using double precision number, an exponent higher than thirteen does not bring any further benefit for the precision.

As stated above, the sine and cosine approximations for the range of $[0, \pi/4]$ are combined to get an approximation over the whole period. Figure 5-5 shows how the different approximations are combined. The cosine and sine approximations are combined as shown previously in Table 5-2. To make the error visible, it is increased by a factor of 40. As reference the blue dashed line represents the cosine. The vertical lines in magenta indicate where a different approximation method is applied. The approximations based on the cosine are printed in red. The value 1.0 is only shown for a x value of 0 and not for 2π. This is in line with the results of chapter 5.1.3, where it is shown that the multiples of π cannot be exactly matched by floating-point numbers. The figure shows an approximation polynomial of the grade two, higher grades would match the curve more often than three times.

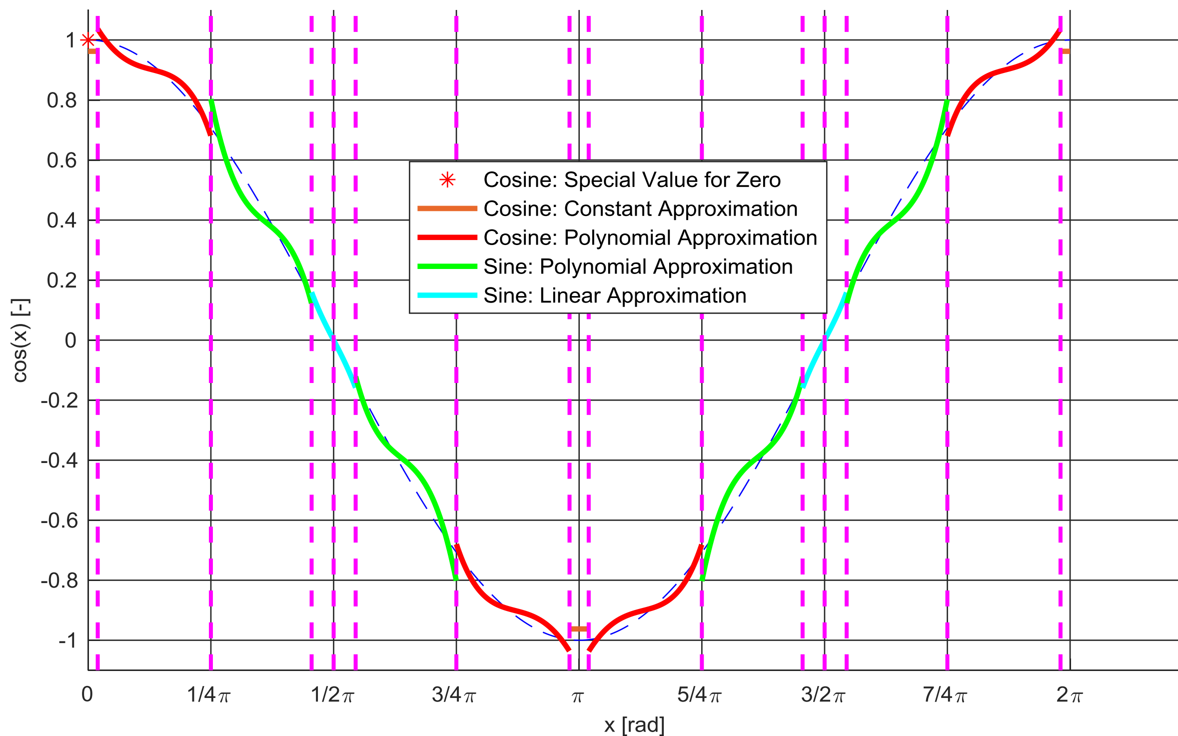

Figure 5-5: Visualization of the Different Approximations

Cosine approximation for the interval $[0, \pi/2]$.

Also, the approximation over the range from 0 to $\pi/2$ with a single polynomial shall be analyzed. In difference to the variants before, here only one polynomial approximation is needed and mapping the whole input range to the range of the core algorithm needs less effort. Therefore, this variant will save space in the memory and potentially has a better WCET. Similar to the approximations before, for input values close to zero and $\pi/2$ the function value is approximated by a constant approximation and a linear approximation. So in difference to the approximations ranging from 0 to $\pi/4$, where only the low bound of the polynomial approximation is adapted, here both the upper and the lower bound have to be adapted. This is also done by a Sollya script. In this case, the bounds do not always converge to a double precision number. To solve this problem, the loop adapting the bounds of the polynomial is limited to maximum of 100 iterations. This limit showed that the bounds already have a very small range in which they vary.

Similar to the cosine approximation before here also two variants of polynomials are analyzed, one with all exponents and one with just the even exponents. The highest exponent for the variant including all integer exponents ranges from two to eleven. In case the rounding errors during the evaluation using double precision are considered, the overall precision is not further increased for exponents bigger than ten. For the second variant with the even exponents, the range of the highest exponent is in a range from two to fourteen. Here, the total error is not further increased for polynomials with a higher grade than twelve. The grades of polynomials are similar to those of the variants where only the range $[0, \pi/4]$ is approximated. But the convergence of the algorithms is not such high as for the other variants.

5.1.5 Algorithm Selection and Implementation Considerations

This section compares the results of the different approximation algorithms and selects a specific variant for the single and double version of the cosine function. Main selection criteria

are the reached precision and the WCET reached by the function. Figure 5-6 shows the precisions for all evaluated variants. Shown is the reached precision for an evaluation using double precision computation over the value of the highest exponent in the polynomial. The additional error introduced by the range reduction is not considered, as it has the same impact on all variants.

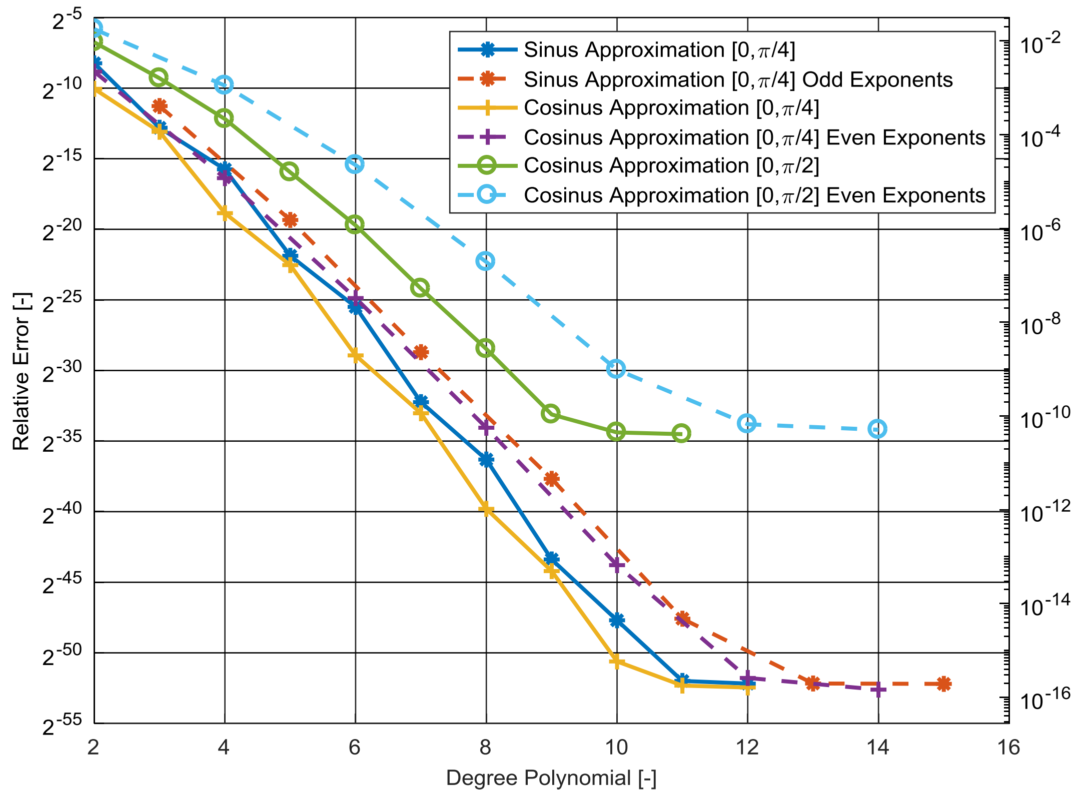

Figure 5-6: Relative Error of the Different Sine / Cosine Approximations

For all variants, the approximation which uses only odd or even exponents, has a slightly worse precision than the one which uses all integer exponents, with the same polynomial degree. The maximum precision which is reached is almost identical for the variant with all exponents and the one with only selected exponents. Keeping in mind that for the evaluation of the variant with the selected exponents the number of floating-point operations is almost half, these might be much more attractive in the context of execution time. The graph also shows that the achieved precision is much lower for the variant approximating the range $[0, \pi/2]$. The reached precision is higher than the precision of a single number. Therefore, for an approximation of the single variant this variant is sufficient. But the variant approximating the range $[0, \pi/2]$ is not able to reach precision close to the double precision. To evaluate, which approximation range is better for the single variant, the WCET aspect needs to be taken into consideration.

In order to show the timing aspects on the applicable target hardware, all variants were implemented in C source code, compiled and analyzed with aiT. If an increased degree of the polynomial did not bring a bigger benefit than one bit in precision, the variant was not transcoded to C. In order to consider the fact that the range reduction of the variant approximating the range $[0, \pi/2]$ is less complex, the WCET also incorporate the range reduction. As described in chapter 9, the library functions are locked to the cache. Therefore, for the analysis of the different variants the option “always hit” [53] is selected for the cache behavior. The polynomials are implemented using the Horner scheme to utilize the benefits of the FMA instruction. This is beneficial for the precision as one intermediate rounding is omitted.

In addition, instead of one add and one multiply only one instruction is necessary. This is beneficial for the code size which is of interest as the cache size is limited. And for the hardware at hand it is also beneficial for the execution time.

For the variant where the range $[0, \pi/2]$ is approximated by one polynomial, the implementation consists of two functions. The first one performs the range reduction and mapping to each quadrant according to section 5.1.4 with a simple "if" statement. After the range reduction the second function is called. This function then performs the actual calculation of the function value, by considering the special cases close to zero and close to $\pi/2$. The reason for the separation of the approximation in an extra function is that this function would also be used by the sine function. So separating the actual approximation to a separate function can save memory.

In order to get a result for the desired range with the polynomials approximating the range $[0, \pi/4]$, both the sine and cosine approximations have to be used. Therefore, here one C function first reduces the input argument and maps the reduced argument to the corresponding approximation afterwards. The mapping according to section 5.1.4 is implemented with a "switch case" statement based on modulo four of the factor of the range reduction. In each case either the sine or cosine approximation functions are called. Both functions perform either the calculation of the polynomial or the special approximation close to zero. The degree of the approximating polynomial is the same for the joined sine and cosine in the case that all integer arguments are used. For the cases with selected exponents, the sine polynomial with the degree n+1 was joined with the cosine polynomial of degree n as these are more equal in the reached precision than an n-1 for the sine and n for the cosine pair. The benefit of the variant with the selected exponents is that the approximations are actually even and odd functions, like the sine and cosine. This enables some simplification in case the reduced argument is negative, as then it is not necessary to consider only the absolute value and reconstruct the value afterwards.

Figure 5-7 shows the precision of the different variants over the analyzed WCET. For the functions where a sine and cosine approximation is combined, always the worse precision is selected.

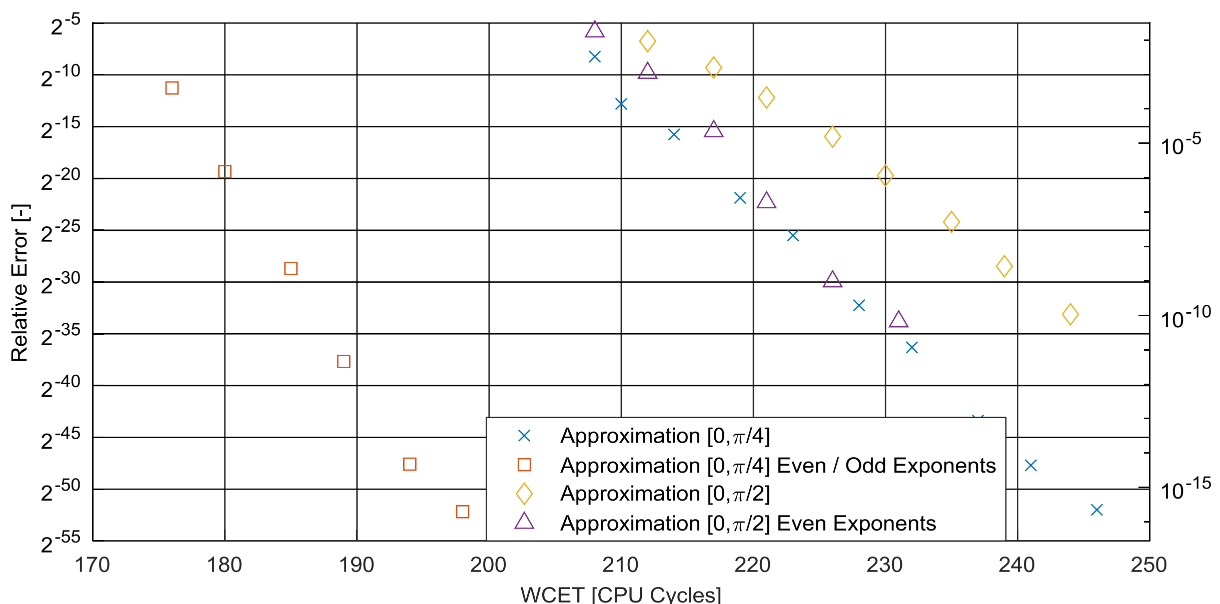

Figure 5-7: WCET and Precision of the Cosine Variants

The figure shows that in order to reach a given precision with the lowest possible WCET, always the variant using an approximation over $[0, \pi/4]$ gives the best result. The reason therefore is that although the degree of the variant with the selected exponents has to be a bit higher to reach the same precision than the variant with all exponents, the necessary instructions are almost half. In addition, due to the odd and even property of the polynomials some parts of the mapping can be eased. The reason why an approximation over the range from $[0, \pi/2]$ does not provide benefits is that the mapping algorithm of the $[0, \pi/4]$ variant is also quite efficient. As the mapping is based on a “switch case” construct with an integer value as input this can be implemented by a branch conditional to count register instruction. So the timing is almost independent of the number of different branches in the mapping algorithm. Therefore, the benefit in the mapping algorithm cannot compensate effort needed for the higher degree polynomial.

Based on these results the single and double variant are based on a polynomial approximation over $[0, \pi/4]$. The following polynomial degrees are selected:

- For the single variant a sine approximation of the degree seven with only odd exponents and a cosine approximation of the degree six with only even exponents
- For the double variant a sine approximation of the degree thirteen with only odd exponents and a cosine approximation of the degree twelve with only even exponents

The total relative error for these variants can be bound to $2^{-24} \approx 6.0 * 10^{-8}$ for the single variant and $2^{-50} \approx 8.9 * 10^{-16}$. For the single variant the error can increase in case the result is rounded back to a single number. Therefore, in this case a precision of $\pm 1ulp$ can be achieved.

In a flight control software, often both the sine and the cosine of an angle are necessary and as both share some common parts one might come to the idea to calculate both values in a single function like [24]. An experimental function calculating both values was implemented. But as the CPU at hand does not really provide a support for a parallel execution of instructions or for SIMD, the execution time benefit of such a function is not very high. So in order to save a bit of memory space, the function was not integrated into the complete software.

5.1.6 Function Stub for the Runtime Error Analysis

In this section the properties used to generate the function stub for the runtime error analysis are presented.

The elementary assumption for both the sine and the cosine algorithm is that the return argument is in the range of $[-1.0, 1.0]$. To proof the absence of the run time errors this assumption is not always enough. For example, an analysis of the model shown in Figure 5-8 will indicate a runtime error if the return range is the only assumption, as the return value of the cosine is potentially zero.

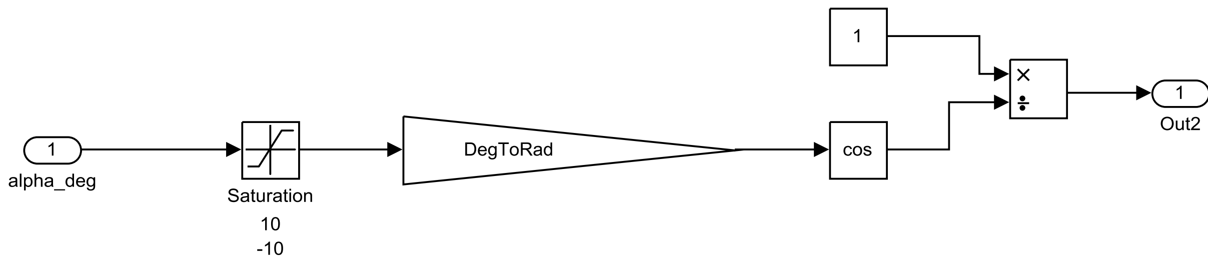

Figure 5-8: Motivation for an Enhanced Cosine Stub

As the input to the cosine is limited in this example, the return value of the cosine cannot get zero. In case of the runtime error analysis this needs to be covered by an additional assumption for the stub. An easy assumption is that for values in the range $[0, \pi/2]$, the cosine is bigger or equal than the linear function crossing the y-axis at one and having a root at $\pi/2$. This assumption can give a lower bound for the cosine function based on the input. The assumption is also simple enough that it can be covered by the method of abstract interpretation. The mathematical expression of this additional boundary condition is:

$$\cos(x) \geq 1 - \frac{2}{\pi} * x \; for \; x \in [0, \frac{\pi}{2}[\tag{5-21}$$

To show that the implementations fulfill the two conditions, all different sections of the approximation must be evaluated: The constant and linear approximation and the two approximation polynomials. In order to check the polynomial, the approximation the Gappa scrips are extended by additional assumptions, which need to be proven. In Gappa y always represents the result of the polynomial calculation executed in double precision. In order to show that the result is always in a range of $[-1{,}0{,}1.0]$ the following assumption is added:

$$|y| <= 1 \tag{5-22}$$

To prove that the approximation is also bigger than the proposed linear lower bound, first equation (5-21) is solved for zero and added as additional statement. This is incorporated as:

```
1  stub_1 IEEEdouble= y - (1-2/3.14159265358979323846 * x);
```

Listing 5-2: Added Statement in Gappa to ease the Proof of the Stub

Based on this statement the following additional assumption is added:

$$stub_1 \geq 0 \tag{5-23}$$

For both additional assumptions Gappa can construct a proof which shows that they are fulfilled. This shows that the stub is compliant with the polynomial approximation.

For the linear approximation, the first assumption that the return value stays in the range $[-1.0{,}1.0]$ is obvious. The linear approximation is only valid for small reduced arguments close to zero and directly returns the input argument or the negative value of it. So for the single variant of the function, where the linear approximation has a bigger range than for the double variant, only inputs where the absolute value is smaller than $1.4 * 10^{-4}$ are subjected to the linear approximation. Therefore, the return range of the linear approximation is [-$1.4 * 10^{-4}, 1.4 * 10^{-4}$], which is obviously contained in $[-1.0{,}1.0]$. That the value is bigger than the linear lower bound can also be shown easily. As the absolute slope of the linear lower bound is approximately 0.64 it is significantly smaller than the absolute slope of the linear

approximation which is one. Due to the correct rounding it is ensured that the result of the linear approximation has at least the same floating-point values than the lower bound. The only problem which could occur is due to rounding the root of the lower bound is actually at a bigger x value than for the cosine function. As shown before, the double representative of $\pi/2$ is smaller than $\pi/2$. The stub in Astrée is implemented using double precision. The actual implementation used a higher precision of $\pi/2$ for the double variant. Therefore, in this case the root of the linear approximation is actually at a bigger value and no problem occurs. The single variant is similar to the stub also based on double precision numbers, so that here also no problem occurs.

For the constant approximation the range statement is easy as the constant approximation has only two return values: Either exactly 1.0 or a value close to 1.0, but a little bit smaller. During the reconstruction to the input argument, the sign of these two values might change. But this is an operation without any rounding, so all values are returned in the interval $[-1.0,1.0]$. Checking against the linear lower bound reveals a problem. As the constant approximation is applied for the floating-point number closest to zero up to the upper bound of the constant approximation, the linear approximation can be bigger than the calculated value. As shown in the following example, this is already the case for the tightest interval where the constant approximation is applied for $x \in]0,2^{-26}]$. In this case, the cosine is approximated by $1 - ulp(1) = 0x3FEFF\ FFFF\ FFFF\ FFFF$. If the assumption of equation (5-21) is evaluated in double precision for the interval from $[0x0,0x3C9921FB54442D18] \approx [0.0,8.72 * 10^{-17}]$ the result will be 1.0. Therefore, for this interval excluding 0.0, the assumption will be wrong. So the assumption must be adapted. This can be easily done by replacing the offset of one with the value of the constant approximation. As this is a more conservative assumption the considerations for the polynomial and the linear part are not affected. Although, it would be possible to adapt the slope, this was not necessary as in cases where the assumption is important the input value had a bigger distance from $\pi/2$. For that reason, the following assumption is correct over the whole range, and solved the underflow problems with cosine function during the runtime error analysis:

$$\cos(x) \geq ConstApprox - \frac{2}{\pi} * x \; for \; x \in [0, \frac{\pi}{2}[\tag{5-24}$$

In addition to the two presented properties, the stub also contains an assertion statement, which checks the input argument for a range of $[-4\pi, 4\pi]$. This is done to show the validity of the design range.

The stub of the sine function is similar. Here also a linear lower bound is used, to give a low bound for some inputs. In this case the linear bound has no offset of zero and a positive slope is applied.

5.1.7 Precision Differential Quotient

The application software in some cases use differential quotients to get a numerical derivative. The precision of such a derivative is also dependent on the precision of the differential quotient of the elementary functions. Therefore, the goal of this section is to bind the absolute error of the differential quotient. The reason why the absolute value is chosen is that the differential quotient itself can get zero. For that reason, a relative error cannot be determined over the whole range.

In general, the differential quotient is defined as follows:

$$\frac{f(x_1) - f(x_2)}{x_1 - x_2} \tag{5-25}$$

Let $\tilde{f}$ in this case be the result of the approximated function. Then the total absolute error is:

$$e_{tot}^{abs} \leq |\tilde{f} - f| \tag{5-26}$$

The absolute error of the differential quotient can be bounded to two times the absolute approximation error for the function divided by delta x.

$$\left|\frac{\tilde{f}(x_1) - \tilde{f}(x_2)}{x_1 - x_2} - \frac{f(x_1) - f(x_2)}{x_1 - x_2}\right| \tag{5-27}$$

$$\left|\frac{f(x_1) \pm e_{tot}^{abs} - f(x_2) \pm e_{tot}^{abs}}{x_1 - x_2} - \frac{f(x_1) - f(x_2)}{x_1 - x_2}\right| \tag{5-28}$$

$$\left|\frac{2 * e_{tot}^{abs}}{x_1 - x_2}\right| \tag{5-29}$$

This is also valid if for the areas where the approximation method is switched as long as the bigger approximation error is used. In this case the error can be reduced by taking once the smaller error and once the bigger error instead of using two times the bigger error. But for a worst case consideration, this fact does not bring any benefit.

The approximations for the sine and cosine were selected based on equality of the relative error over the whole function range. Therefore, the constant approximation close to the function values of 1.0 probably will have the biggest absolute error of the different approximations. To ensure this, the Gappa scripts were extended to also determine the absolute error. The result is given in Figure 5-9. Displayed are the absolute errors of the polynomial approximations and the absolute errors of the associated linear or constant approximation. As the linear and constant approximation have a constant polynomial degree the x axis is named associated polynomial degree.

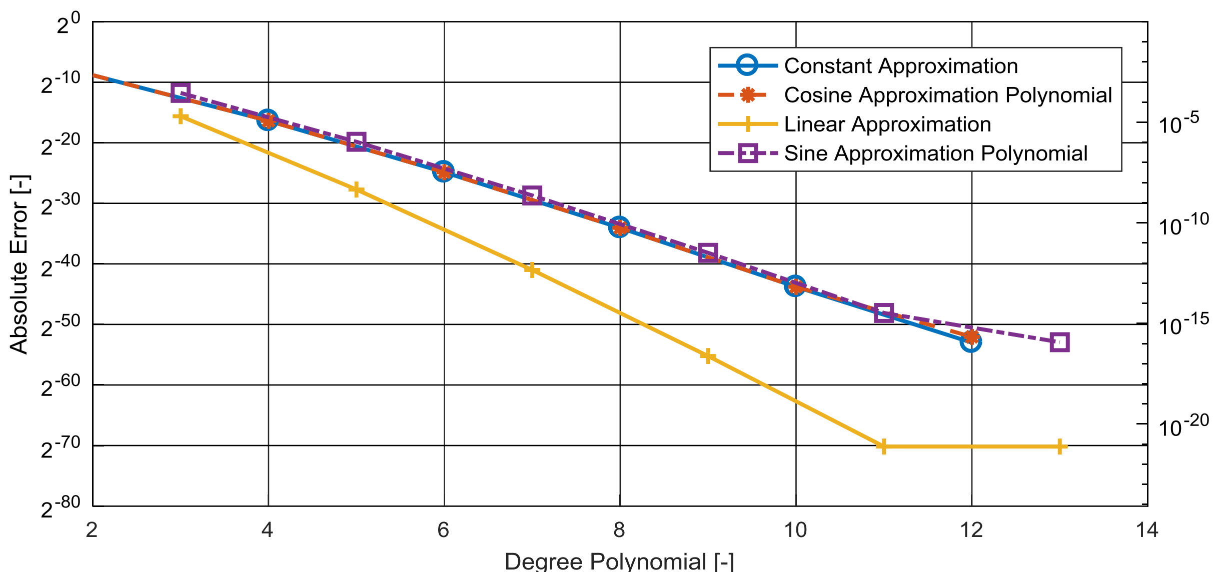

Figure 5-9: Sine and Cosine Absolute Errors

For the polynomial approximations in the figure only the following are shown:

- The ones with only even exponents for the cosine
- The ones with only odd exponents for the sine

In difference to the expectation, the errors for the constant approximation and the two polynomial approximations is almost the same. The reason for this is that all the approximations are at least in some parts valid for return values with an unbiased exponent of minus one. Therefore, a similar relative error also corresponds to a similar absolute error. Only for the linear approximation, which is only valid for numbers with a smaller exponent, the absolute error is smaller than for the rest. For the first selected version with a polynomial grade of six and seven, the absolute error is in the range of $2^{-24} \approx 6.0 * 10^{-8}$. For the higher precision version with a polynomial grade of twelve and thirteen, the absolute error is in arrange of $2^{-48} \approx 3.6 * 10^{-15}$. Both of the values should be small enough to get a meaningful differential quotient in case the base is reasonable selected.

In case this is still not precise enough, the value could be further bounded as described in the next paragraphs. The result of the approximation function evaluated in double precision can be expressed as:

$$\tilde{f}(x) = f(x) + e_{rnd}^{abs}(x) + e_{apr}^{abs}(x) \tag{5-30}$$

For the differential quotient follows:

$$\tilde{\delta} = \frac{f(x_1) + e_{rnd}^{abs}(x_1) + e_{apr}^{abs}(x_1) - f(x_2) - e_{rnd}^{abs}(x_2) - e_{apr}^{abs}(x_2)}{x_1 - x_2} \tag{5-31}$$

$$\tilde{\delta} = \delta + \frac{e_{apr}^{abs}(x_1) - e_{rnd}^{abs}(x_2) + e_{rnd}^{abs}(x_1) - e_{apr}^{abs}(x_2)}{x_1 - x_2} \tag{5-32}$$

The rounding error can be bounded by two times the maximum of the rounding error. To bind the difference of the two approximations the absolute error of the derivative of the approximation is calculated.

$$e_{apr}^{abs^*}(x) = f'_{apr}(x) - f'(x) \tag{5-33}$$

This term can be easily calculated using Sollya. If the maximum value of equation (5-33) has been determined, the difference of the two approximation errors can be calculated as follows:

$$\begin{aligned} \left| e_{apr}^{abs}(x_1) - e_{apr}^{abs}(x_2) \right| &= \left| \int_{x_2}^{x_1} f'_{apr}(x) - f'(x) dx \right| \\ &\leq \max\left(f'_{apr}(x) - f'(x) \right) * (x_1 - x_2) \end{aligned} \tag{5-34}$$

$$\tilde{\delta} \leq \delta + \max\left(f'_{approx}(x) - f'(x) \right) + \frac{2err_{round}}{x_1 - x_2} \tag{5-35}$$

For the given variants of the functions the rounding error dominates. Therefore, the benefit of equation (5-35) is limited and the term also was not determined with Sollya. Anyway, from numerical point of view a differential quotient always is a critical statement, which needs to be handled with care. Due to the limited resolution of floating-point numbers even with a correctly

rounded function, all bits of a numerical quotient can become wrong in case the differential base is too small.

5.2 Tangent

Next to the sine and cosine the tangent function is the third important function dealing with angles in a rectangular triangle. The tangent can be defined as the ratio of the opposite leg to adjacent leg in the rectangle triangle. This is equivalent to the ratio of $\sin(x)/\cos(x)$. As the tangent is based on this ratio the function has a singularity at $\pi/2$. The period of the function is π. As shown in Table 3-1, for the software example at hand the tangent function is not commonly used. The usage is limited to the inner loop, the trajectory generation and the trajectory control module.

5.2.1 Input Restrictions

As specified in the ANSI C [47] the implementation here shall return the tangent of an input argument given in radians. As embedded systems normally try to omit singularities the input range to the tangent is restricted to $]\widetilde{-\pi/2}, \widetilde{\pi/2}[$. This range is also in line with the used flight control software. As the function has a singularity for an input of $\pi/2$ a special handling of this value might be necessary. But as the double representative of $\pi/2$ and the real value of $\pi/2$ are not equal, $\tan(\widetilde{\pi/2})$ is actually defined and the correctly to the nearest rounded result is 16331239353195370. Therefore, no special handling needs to be considered for these input arguments. As the input range is restricted to $\pm\widetilde{\pi/2}$ it is not necessary to check the cases where the distance between the integer multiples of $\pi/2$ and their double representative are smaller. The smaller distance to the singularity might lead to an overflow in those cases. For the single variant it has to be considered that the single number closest to $\pi/2$ is actually bigger than $\pi/2$. For that reason, the limits here have to be defined as the single representative of $\pm\pi/2$ rounded towards zero.

5.2.2 Architecture

This subsection discusses different approaches that can be utilized to implement the tangent function. Based on this overview the variants, which then are further explored in the next subsections are selected. In the “crlibm” the tangent is calculated by a division of the sine and cosine [27]. In this case the approximation errors of both functions might amplify. In case of the high precision of the “crlibm” this is not a problem but the approximation here is not such precise, so the division is not an option. In addition, prior to the division both the sine and cosine need to be calculated, which leads to a significant computational effort. Therefore, the WCET of a routine designed in such way might not be optimal. [11] proposes for the calculation of the tangent a range reduction to $[0, \alpha\pi]$, with $\alpha \in \left\{\frac{1}{4}, \frac{1}{8}, \frac{1}{12}, \frac{1}{16}\right\}$. To reconstruct the return argument for the initial value the following formals are proposed:

$$\tan(x) = \frac{1}{\tan\left(\frac{\pi}{2} - x\right)} \tag{5-36}$$

$$\tan(x) = \frac{\tan(a) + \tan(x - a)}{1 - \tan(a)\tan(x - a)} \tag{5-37}$$

$$\tan(x) = \frac{2\tan(x/2)}{1 - \tan^2(x/2)} \tag{5-38}$$

For the implementation here the input range is restricted to $[-\widetilde{\pi/2}, \widetilde{\pi/2}]$. The runtime error analysis shows that this bound is not violated. In order to give a good approximation on this range, two variants are evaluated here. Without loss of generality the following argumentation only considers positive inputs. The first variant reduces only input arguments bigger than $\pi/4$ by subtracting the input argument from $\pi/2$. For the reconstruction of the argument equation (5-36) is used. Values smaller than $\pi/4$ are directly approximated by a polynomial. For the second variant the polynomial approximates only the small range up to $\pi/32$. The range is reduced using an additive range reduction, similar to the sine and cosine but based on $\pi/16$. In order to reconstruct the return value, equation (5-37) is used. To execute this calculation, additionally the values of $\tan\left(\frac{\widetilde{n\pi}}{16}\right)$ *with* $n \in [1,2,\ldots,8]$ are stored. Important is that not the tangent of the actual multiple of $\pi/16$ is stored but the tangent of the double representative of the multiple. As equation (5-37) is not restricted to certain multiples of π this is possible and eases the range reduction (Compare 5.2.3). The approach also enables the calculation of equation (5-37) for the limit case that the input is equal to $\widetilde{\pi/2}$. The other variants with $\alpha = \frac{1}{8}, \frac{1}{12}, \frac{1}{16}$ are not evaluated because the necessary computation steps for the range reduction are the same as the variant with $\alpha = 1/32$. And due to the bigger range of the polynomial approximation, the degree of the polynomial will increase in case the same precision shall be reached.

Figure 5-10 shows a comparison of the variant with a polynomial approximation for the range of $[0, \pi/32]$ and for range of $[0, \pi/4]$. For both variants, polynomials with full exponents and only odd exponents are shown. Similar to the sine, the variants that contain only odd exponents have almost the same precision as the variants with all exponents. Therefore, due to the execution time aspect, in the following only the variants with odd exponents are considered. As expected, the variant approximating values up to $\pi/32$ has a faster convergence but the range reduction is a bit more complex. Interesting is also, that for the variant approximating values up to $\pi/4$ not always the best approximation is returned. As for the grades 19 to 22 the performance is lower than for the one with a grade of 18.

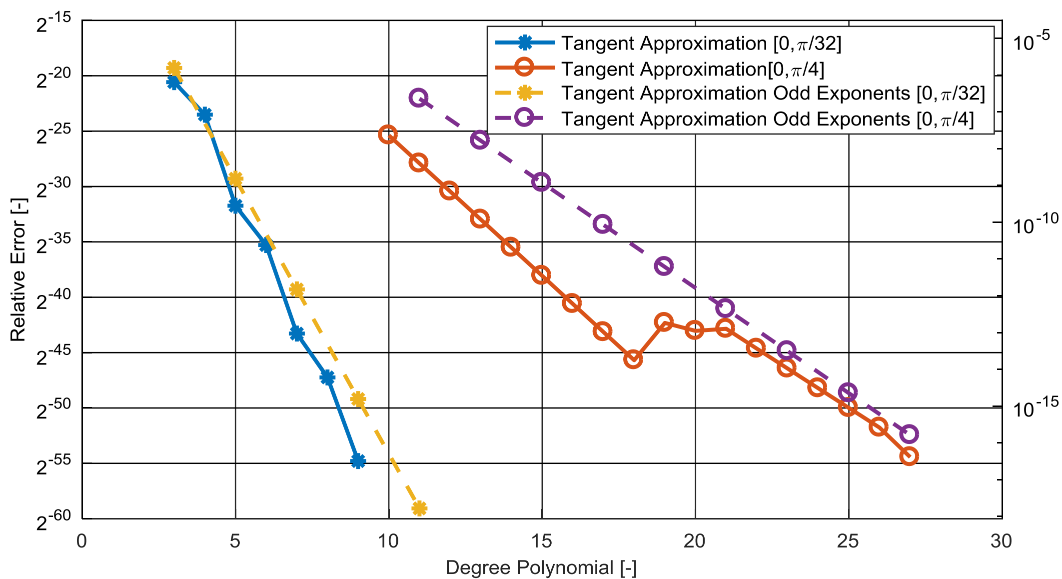

Figure 5-10: Precision of Different Tangent Approximation Polynomials

5.2.3 Range Reduction

As stated above, the variant with and approximation up to $\pi/4$ uses a range reduction based on equation (5-36). In order to reduce the range, the input arguments to the tangent function bigger than $\pi/4$ must be subtracted from $\pi/2$:

$$x_{red} = \frac{\pi}{2} - x \tag{5-39}$$

As x can get close to $\pi/2$ the result of equation might be subject to cancelation if it is executed in normal double precision. In order to omit this problem, the constant $\pi/2$ is represented and stored by two double precision numbers. Therefore, the subtraction of equation (5-39) is implemented in the source code by a subtraction and a sum, as shown in Listing 5-3:

```
1  x_red = HALF_PI - x_abs;
2  x_red = x_red + HALF_PI_DD;
```

Listing 5-3: Two Step Tangent Range Reduction

As stated above the input argument of the tangent is smaller or equal to the double representative of $\pi/2$. So the lower limit of the reduced argument is bound by:

$$x_{red} \geq \frac{\pi}{2} - \frac{\tilde{\pi}}{2} \approx 6.12 * 10^{-17} \tag{5-40}$$

The maximum of the relative error of the range reduction is calculated as follows:

$$max\left(e^{rel}\right) = \left| \frac{\left(\frac{\pi}{2} - \left(\frac{\tilde{\pi}}{2} + \widetilde{\frac{\pi}{2} - \frac{\tilde{\pi}}{2}}\right)\right)}{\frac{\pi}{2} - \frac{\tilde{\pi}}{2}} \right| \tag{5-41}$$

The enumerator is the absolute error of the approximation of $\pi/2$ by two double precision numbers. The value is approximately $1.5 * 10^{-33} < 2^{-109}$. The denominator is the difference between $\pi/2$ and the input value which is closest to $\pi/2$, so in this case the double representative of it. For that reason, the denominator approximately resolves to: $6.12 * 10^{-17} > 2^{-54}$

So the relative error can be bound to:

$$e^{rel} \leq \frac{2^{-109}}{2^{-54}} = 2^{-55} \tag{5-42}$$

As the result of the range reduction is stored using a double precision number, the rounding error must also be considered so that $e^{rel} \leq 2^{-52}$.

For the variant where only the range up to $\pi/32$ is approximated by the polynomials, the reconstruction is based on equation (5-37). Therefore, in the term $a - x$, a must be selected in a way that the result is in the range $[-\pi/32, \pi/ 32]$. This can be achieved by setting a to $n * \pi/16$. But the equation to reconstruct the full range result is not necessarily based on an integer multiple of $\pi/2^n$, with n being an integer number. This fact can be utilized by the range reduction, using the double representative of $\pi/16$ instead of the real number, omits the need of a representable with a higher precision. If only the double number is used during the range

reduction, for the reconstruction to the full input range instead of the value $\tan(n * \pi/16)$, $\tan(n * \widetilde{\pi/16})$ needs to be used for the term $\tan(a)$.

In order to get the factor n in the first step of the range reduction, the absolute of input argument is multiplied by the inverse of $\pi/16$ and the result is rounded to the nearest integer. As second step the result is multiplied with $\widetilde{\pi/16}$ and then subtracted from the absolute value of the input. The result is the reduced argument in the range of $[-\pi/32, \pi/32]$.

In terms of necessary precision, for the first step of the range reduction it is sufficient if the correct rounded integer is returned. If the absolute error is smaller then 0.5, this is the case. As the calculation is executed using a FMA instruction this is fulfilled for all arguments within the design range. The implementation of this step consists out of a FMA instruction and a cast. First, the multiplication with the inverse of $\widetilde{\pi/16}$ is executed. And in order to get an integer value, the cast is executed. That the value is not truncated and rounded to the nearest integer instead, the value of 0.5 is added in advance to the cast.

```
1  factor = (CPU_INT08S)(cpu_fma(PI16_inverse,x_abs,0.5));
```

Listing 5-4: Calculation of the Factor for the Additive Tangent Range Reduction

As the calculation is implemented using a FMA instruction, it is possible to proof that the calculation is correctly rounded for the complete range of the input `x_abs` from 0 to $\widetilde{\pi/2}$. This leads to the fact that `factor` is in the range of $[0{,}8]$. To show the precision of the second step, the fact that the three LSBs of $\widetilde{\pi/16}$ are zero is used. It follows that the following equation is valid for all integer n in the range $[0{,}8]$:

$$\frac{\tilde{\pi}}{16} \times n = \frac{\tilde{\pi}}{16} \otimes n \tag{5-43}$$

So the multiplication does not introduce a rounding error. As this is true for $\widetilde{\pi/16}$, the variant with an polynomial approximation from $\pm\pi/32$ was added in addition to the versions proposed in [11]. Equation (5-43) shows the calculation of the variable a in equation (5-37). In order to get the reduced argument a has to be subtracted from the input argument. To show the precision of this subtraction Sterbenz lemma, as introduced in section 2.2, can be used. To proof the validity in this case, it must be shown that the following condition is fulfilled:

$$\frac{|x|}{2} \leq \frac{\tilde{\pi}}{16} \otimes n \leq 2 \times |x| \tag{5-44}$$

Where n is equal to `factor` which correlates to $\lfloor |x| \times 16/\pi \rceil$ and $|x|$ is the absolute value of the input. So for the input range $[0, \pi/2]$ for $|x|$ it is easy to see that $|x| = \pi/32$ is the most critical case in order to fulfill equation (5-44). In this case n gets to one but as $\widetilde{\pi/32}$ is exactly the half of $\widetilde{\pi/16}$ Sterbenz lemma is still valid. Also, for the case that $|x| = \frac{\pi}{32} - ulp\left(\frac{\pi}{32}\right)$ there occurs no problem as `factor` resolves to zero in this case and $x_{red} = |x|$. So in case of the additive reduction to $x_{red} \in [-\pi/32, \pi/32]$ no error is introduced.

5.2.4 Core Algorithm

Both variants of the core algorithm use a linear approximation with a slope of one for arguments close to zero and a higher odd polynomial approximation for arguments where the absolute value is above a certain limit. The polynomials for both variants are implemented in Horner form using FMA instructions, in order to take benefit of the faster execution and the higher

intermediate precision. The bound of the linear approximation is determined using the find zeros command of Sollya. The code below shows how to calculate the maximum bound of the linear approximation close to zero which still provides double precision.

```
1  a = findzeros(tan(x)*(1-2^-53)-(x),[2^-100,1]);
```

Listing 5-5: Sollya Code to Determine the Upper Bound of the Linear Approximation

As for positive numbers only one root exists, there is no need to pay attention on the selection of the correct solution. But in order to get useful results, the diam parameter of Sollya must be set to a smaller value than the default one. The diam parameter controls the minimum size of the subintervals, which are internally used to solve the problem in a safe manner (Compare section 4.1). The statement returns $a \approx 2^{-25.7}$, so for values below 2^{-26} the linear approximation returns a result with double precision. Similar to the sine and cosine implementation, the bound should be adapted to be in line with the precision of the polynomial. Therefore, the code snippet above is also included in the loop which determines the polynomial approximation but the hard coded 2^-53 is replaced by the relative error of the polynomial approximation.

Next to the different range, the main difference between the polynomial approximations is that in case of the approximation up to $\pi/4$ the range reduction returns only positive values as in the other case also negative values can be returned. As both the polynomial and the tangent are odd functions, the sign does not matter and there is no need to calculate the absolute value. Only the check if the linear approximation is applicable must consider that the value also can be negative. In order to omit an additional `fabs` instruction, the squared reduced input argument is used. This is always positive and so the check if the argument is close to zero can be done with a simple comparison to the squared bound. The squared input argument is anyway needed for the polynomial calculation and as the linear approximation is not on the WCET path the WCET is not increased if the square of the input argument is calculated before the branch. The squared bound is calculated offline and stored instead of the direct bound.

The Gappa script for both polynomials is straightforward and are exactly orientated on the approach described in 4.1. The script already considers the applicable errors of the range reduction according to section 5.2.3. Table 5-3 shows the precision reached by the polynomial approximation of selected variants of the tangent.

Variant	Approximation Range	Degree Polynomial	Relative Error of Approximation
`tan_pi4_13`	$[0, \pi/4]$	13	$e_{apr}^{rel} = 2^{-25.8}$
`tan_pi4_27`	$[0, \pi/4]$	27	$e_{apr}^{rel} = 2^{-50.1}$
`tan_pi32_5`	$[0, \pi/32]$	5	$e_{apr}^{rel} = 2^{-29.2}$
`tan_pi32_11`	$[0, \pi/32]$	11	$e_{apr}^{rel} = 2^{-51.9}$

Table 5-3: Precisions Reached by the Polynomial Approximations of the Tangent

5.2.5 Argument Reconstruction

The reconstruction of the sign of the result is for both variants the same, based on the flag, calculated as described in section 4.2, the return value is negated or not. This does not introduce an additional error.

In case that the core algorithm provides an approximation up to $\pi/4$ for the reconstruction of inputs bigger than $\pi/4$ equation (5-36) is used. This equation can be implemented by a normal division instruction, so the error introduced by the reconstruction can be bound using the

correctly rounding property of instructions defined in IEEE 754 [3]. As the relative errors of the polynomial approximations are bigger than the relative error of a correctly rounded instruction, the results of Table 5-3 keep their validity for the whole range.

The result of equation (5-40) can be used to show that the argument does not overflow. The minimum result of the range reduction is between 2^{-54} and 2^{-53}. Which is again smaller than the lowest bound of the linear approximation 2^{-26} (compare 5.2.4). Therefore, for values close to $\pi/2$ the linear approximation is applicable and the upper bound of the return argument of the tangent can be bound by the inverse of the reduced argument in equation (5-40).

$$\tan(x) \leq \frac{1}{x_{red}} \leq \frac{1}{2^{-54}} * (1 + 2^{-53}) = 2^{54} + 2 = 18014398509481986 \tag{5-45}$$

This is much smaller than the biggest representable double number. For that reason, the tangent does not overflow. The value is a conservative estimation for the upper bound. So, the value is bigger as the biggest correctly rounded result given in section 5.2.1. As the reduced argument in the actual implementation is bigger than 2^{-54} the value given in the equation above is actually not reached here.

For the variant where the core algorithm approximates ranges up to $\pi/32$ equation (5-37) is used for the reconstruction of the return value for input values bigger than $\pi/32$. The values for $\tan(a)$ are all stored in one interval. The applicable value is selected by setting the index of the interval to the value of the `factor` determined during the range reduction. In order to calculate the denominator a fused negative multiply add instruction is used. The enumerator is calculated by a normal add and also the division is implemented using the standard instruction. In order to show that achieved precision, Gappa scripts are compiled for each sub-interval of the range reduction. The Gappa basically contain the implementation of equation (5-37) twice. Once as transcript of the source code with double precision calculations and once as mathematical definition without rounding operators. The mathematical definition also works with the actual values of the offset $\tan(a)$ and for the tangent on the reduced range $\tan(x - a)$. In order to provide a successful meaningful proof, a correlation between these actual values and their double representatives must be defined. The relative error of the term $\tan(a)$ can be bound by 2^{-53}, as the correctly rounded to the nearest double representative is stored. For the $\tan(x - a)$ the absolute and relative error of the proof for the core algorithm is used. Additional to these errors, the value of $\tan(a)$ and the ranges of the approximation and the overall results are given. In some cases the zero case is excluded from the range of the polynomial approximation. This is done by specifying the range of the absolute value of the polynomial approximation result. The benefit is that the lower bound can be increased by one $ulp(\mathrm{a})$. Consequently, the proof must not deal with de-normalized numbers. The cases where the input value is exactly equal to a bound and so returns a correctly rounded result is analog to the explanation below. With these considerations, the Gappa scripts are sufficient to proof the precision for all intervals with exception for the last interval. For the statement handling the biggest inputs, the limit case that the input to the tangent is equal to $\widetilde{\pi/2}$ is handled manually.

For $x = \widetilde{\pi/2}$ $x - a$ resolves to an exact zero and as the approximation for the tangent of zero is exact $\tan(x - a)$ also resolves to zero. Therefore, the reconstruction is as follows:

$$\tan\left(\frac{\tilde{\pi}}{2}\right) = \left(\widetilde{\tan\left(\frac{\tilde{\pi}}{2}\right)} \oplus 0.0\right) \oslash \left(1.0 \ominus \widetilde{\tan\left(\frac{\tilde{\pi}}{2}\right)} \otimes 0.0\right) \tag{5-46}$$

As $\widetilde{\tan\left(\frac{\pi}{2}\right)}$ is non infinite, the addition and the multiplication with zero is exact. Thus resolves (5-46) to:

$$\tan\left(\frac{\widetilde{\pi}}{2}\right) = \left(\widetilde{\tan\left(\frac{\widetilde{\pi}}{2}\right)}\right) \oslash 1.0 = \widetilde{\tan\left(\frac{\widetilde{\pi}}{2}\right)} \tag{5-47}$$

As the division with 1.0 is also exact, in the limit case exactly the correctly rounded result is returned. So, the relative error in this case is $e_{tot}^{rel} \leq 2^{-53}$.

For the rest of the interval again a Gappa script is used. The difference to the other scripts is that the range of the mathematical definition of the tangent on the reduced range is limited to negative values. As the zero case is handled above manually, the upper limit of the input to the polynomial approximation here is $ulp(\pi/2)$. As the tangent is a strictly increasing function the upper limit can be set as follows:

$$\tan\left(-ulp\left(\frac{\pi}{2}\right)\right) < 2^{-52} \tag{5-48}$$

Table 5-4 shows the precision which could be proven for the whole range:

Variant	Total Error
`tan_pi32_5`	$e_{tot}^{rel} = 2^{-28.2}$
`tan_pi32_11`	$e_{tot}^{rel} = 2^{-49.1}$

Table 5-4: Precisions Reached for the Complete Input Range for `tan_pi32` Variants

Additionally, the range is checked for an overflow. Again equation (5-37) is used. In case the input argument to the tangent is in the range of $]15\pi/32, 16\pi/32[$, a is equal to $\widetilde{\pi/2}$. Therefore: $x - a < 0.0$ and $\widetilde{\tan(x-a)} < 0.0$. So for the enumerator of equation (5-37) the following is valid in this case:

$$\tan(a) \oplus \tan(x-a) \leq \tan(a) \tag{5-49}$$

And the following for the denominator:

$$1 \ominus \tan(a) \otimes \tan(x-a) \geq 1 \tag{5-50}$$

Consequently, in this case $\tan(x) \leq \tan(a)$. As stated above for the limit case that the input is exactly $\widetilde{\pi/2}$ the result is the correctly rounded to the nearest of $\tan(\widetilde{\pi/2})$. So, the absolute of the return argument of the tangent function in this case can be bound by:

$$|\tan(x)| \leq \widetilde{\tan\left(\frac{\widetilde{\pi}}{2}\right)} = 16331239353195370 \tag{5-51}$$

5.2.6 Algorithm Selection

Without an actual implementation it is difficult to determine if the WCET of the variant with a polynomial approximation up to $\pi/4$ or the one with the polynomial approximation up to $\pi/32$ will be bigger. The one with the smaller approximation range has a more complicated range reduction and argument reconstruction but the degree of the polynomial can be small. For the variant with the bigger approximation range it is vice versa. The range reduction and reconstruction are quite simple in case the polynomial has a higher degree. Therefore, there

is a tradeoff between those variants which cannot be easily decided. So both variants are implemented and a WCET analysis is performed using aiT. The results are shown in Figure 5-11:

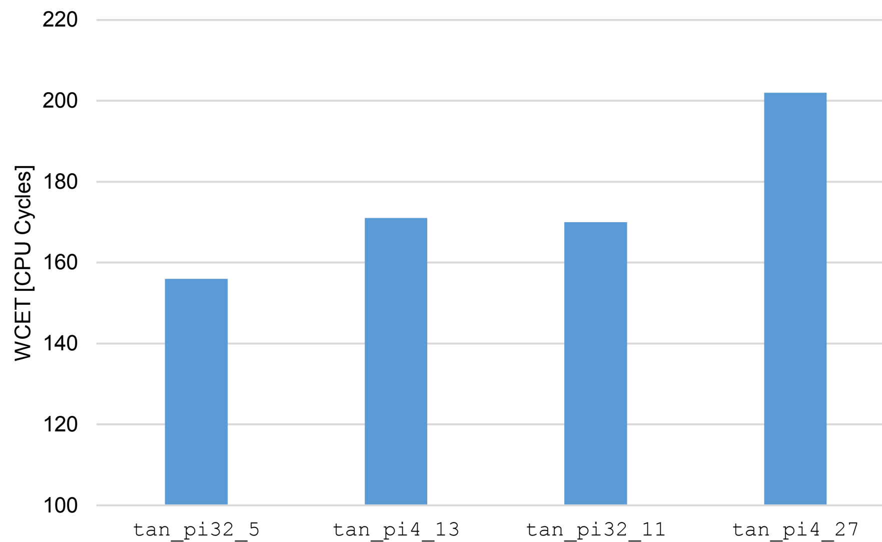

Figure 5-11: Analyzed WCET for the Tangent Function

The `tan_pi32_5`, and `tan_pi4_13` both reach a precision of $\pm ulp(\tilde{x}^{single})$. For that reason, these variants are the preferred versions when tanf is called in the code. The other two variants reach almost double precision. Independent of the desired precision, the variant using a polynomial approximation up to $\pi/32$ is faster. Hence this variant is selected for the integration into the flight control software.

5.2.7 Function Stub for the Runtime Error Analysis

The stub created for the tangent function first checks the validity of the input range. For the software at hand it was possible to restrict the range of the assert statement even further. Instead of applying the design range of $\pm\widetilde{\pi/2}$, a limit of $\pm 89° = \pm 1.5533430342749535$ could be shown.

For the output the simplest stub was sufficient. This is the limitation that the return value stays within the maxima values. This maximum value is given in equation (5-51). As this is only the positive maximum, it has to be extended to the negative minimum. The negative value has the same absolute value only a different sign. Consequently, the output range is restricted to $[-16331239353195370, 16331239353195370]$.

In case for other applications this is not sufficient, the sub can be in improved by inserting statements checking for a more limited input range. In case this range is smaller, the monolithically increasing property of the tangent can be used and the return value can be limited to the tangent of the reduced limit values. In order to ensure a sound stubbing, in this case the return value shall be increased by a multiplication with $(1 + e_{tot}^{rel})$.

5.3 Arctangent

The arcus tangent is the inverse function of the tangent. The function is defined for all input numbers and has no singularities. The range of the return values is $]-\pi/2, \pi/2[$. This means

that all returned angles only represent the quadrant one and four of the unit circle shown in Figure 5-1. In order to get also results in the quadrant two and three a two argument version of the arcus tangent is defined. Instead of passing directly the result of a ration to the function, both components are passed separately to the function. Based on the combination of the sign and the ratio of the components the result can then be allocated to one of the four quadrants. As shown in Table 3-1 the arcus tangent `atan` and the two argument version `atan2` are called overall at 52 call sites. From these 48 are on the analyzed WCET path of the software at hand.

5.3.1 Input Restrictions

As the arcus tangent is defined for all rational numbers, the input range to this function does not need to be restricted. As defined in the ANSI C [47] the return argument is an angle in radians. For the `atan2` the first argument is the numerator, and the second argument is the denominator. In order to omit a division by zero, the second argument must be checked for being zero. If the second argument is zero, the return value is based on the sign of the first input either zero or π. ANSI C defines: In case both input arguments are zero, a domain error could be raised [47]. In difference to this behavior the IEEE 754 [3] defines the following: $atan2(\pm 0, -0)$ is $\pm\pi$ and $atan2(\pm 0, +0)$ is ± 0. As both behaviors are not mandatory, in an experiment the behavior of the `atan2` function in a Simulink simulation is checked. Here the `atan2` returns zero in case both arguments are zero. The zero is returned independent of the sign of the zero input. The behavior, that for all cases, where both inputs are zero, a zero is returned, shall be incorporated in the implemented function. This variant is chosen to match the default behavior of Simulink.

5.3.2 Architecture

The design of the arcus tangent function is as proposed by [11]. The input range is reduced using the following equalities:

$$arctan(x) = arctan(x_i) - arctan(t) \tag{5-52}$$

With t equal to:

$$t = \frac{x - x_i}{1 + x * x_i} = x_i^{-1} - \frac{x_i^{-2} + 1}{x_i^{-1} + x} \tag{5-53}$$

[11] proposes to switch the offset $arctan(x_i)$ at the points X_i. The calculations of X_i is shown in equations (5-54) to (5-56). In order to correlate each X_i with the corresponding x_i, which is necessary to calculate t the additional integer variable i is introduced.

$$X_0 = 0 \tag{5-54}$$

$$X_i = \tan\left[\frac{(2i - 1)\pi}{4s}\right] \; with \; i = 1, \dots, s \tag{5-55}$$

$$X_{s+1} = \infty \tag{5-56}$$

$$x_i = \tan\left[\frac{(2i - 2)\pi}{4s}\right] \; with \; i = 2, \dots, s + 1 \tag{5-57}$$

For a better understanding Figure 5-12 gives a visualization of the relevant parameters for $s = 3$. In blue the arcus tangent function is shown. The bounds of the approximation ranges, limited

by X_i, are given by the red dotted lines. And the relevant offsets $\mathrm{atan}(x_i)$ of range reduction operator x_i are shown as red circles.

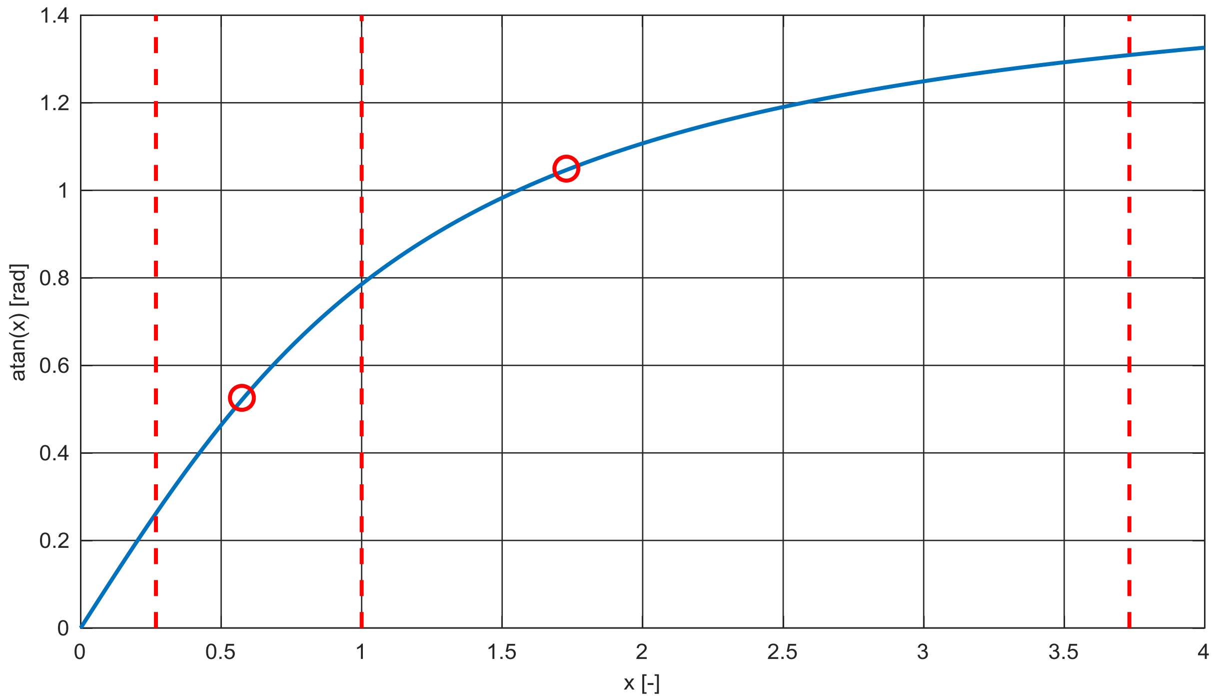

Figure 5-12: Visualization of the Arcus Tangent Approximation Intervals

[11] lists polynomials for: $s = 1,2,3,4, and\ 8$. Here polynomials are generated using Sollya for $s = 2,3,4,5,7, and\ 8$. The value of $s = 1$ is not considered as here the degree of the approximation polynomial would be quite high. The value of seven was added, as the range reduction will be efficient in this case. All paths in the binary search tree will have the same length. Therefore, for a maximum path length of the search the size of the polynomial approximation interval is minimal. For the calculation of the range reduction not the second part of equation (5-53) is used. In order to enable a fast calculation in this case $1/x_i$ and $1 - 1/x_i^2$ need to be stored. If the first part is used only x_i must be stored and the denominator can be calculated using a FMA instruction. For the last interval the following is valid:

$$x_i = x_{s+1} = \infty \tag{5-58}$$

Therefore, the first part of equation (5-53) cannot be calculated as the term $x * x_i$ would overflow and the term $x - x_i$ would underflow. As consequence, the second part of the equation (5-53) is used. The terms x_i^{-1} and x_i^{-2} both resolve to zero for the case that $x_i = \infty$.This leads to the simplification of equation (5-53), valid for the last interval:

$$t = -\frac{1}{x} \tag{5-59}$$

5.3.3 Range Reduction

The arcus tangent is approximated by a polynomial for the range $[0, X_1]$. Other input arguments are reduced using equation (5-53) to the interval $[-X_1, X_1]$. As the breakpoints X_i are not equally distributed, a binary search is performed to determine the applicable values for x_i and $\arctan(x_i)$. Therefore, the computation time of the range reduction will increase with bigger

values of s. But as the range where the arcus tangent is approximated by a polynomial decreases with bigger values for s, the grade of the polynomial probably can be lower in order to reach a given precision.

To show the overall result, first the error of the range reduction must be determined, then this error can be provided in the script for the polynomial approximation. Due to the deviation between the reduced Mx and x, a slightly bigger error for the polynomial approximation is expected. In order to get the final error this error must be considered for the final adjustment step.

In the following only the variant using $s = 3$, which results in the best WCET, is considered. For the other variants the argumentation is similar.

As stated above the first part of equation (5-53) is used to reduce the range for the two middle part intervals. Here the enumerator can get zero and so t also can get zero. Thus, a calculation of a relative error is not possible. So, for the range reduction in this intervals, only the absolute error is considered and afterwards forwarded to the error calculation of the polynomial evaluation by bounding the expression $|x - Mx|$. As the return value of the arcus tangent is not getting zero for input values belonging to this interval, only determining the absolute error is sufficient. For binding the total error to a tight bound, it is also relevant that errors are only considered once and at the right place. As stated in equation (5-53), x_i, the tangent of a fraction of π, is used to reduce the input argument. As this number is not exactly representable in double precision, the first approximation error normally needs to be considered here. As the equation is not restricted to certain fractions of π, the double representation of x_i is used to reduce the argument. Consequently, in the range reduction no approximation of the constants has to be considered and the Gappa script is a transcript of the C code and the mathematical definition. In order to make no additional rounding error in the final reconstruction step, the offset $\operatorname{atan}(x_i)$ has to be calculated as the arcus tangent of the double representation x_i and not of the real number $(2i-2)\pi/4s$. Because x_i is chosen such that it is a double number, Sterbenz lemma can also be used over huge parts of the intervals to further reduce the error of the range reduction. For the parts where Sterbenz lemma is fulfilled, the enumerator of the first part of equation (5-53) can be calculated without error. The case the enumerator gets zero can be excluded, as there no error in the range reduction is made. As shown in equation (5-59), the range reduction in the last interval is only a division, the correct rounding property of basic functions can be used to bound the relative error to $e^{rel} \leq 2^{-53}$.

5.3.4 Core Algorithm

The core algorithm approximates the arcus tangent for input arguments in the range of $[0, X_1]$. As the range reduction can return negative arguments it would be better if the core algorithm also is an odd function that the range of [-X_1,X_1] can be approximated easily. Similar to the tangent here also arguments close to zero are approximated by a linear approximation with the slope of one. As the inverse of such a linear function is the function itself, the argumentation to show the validity of this approximation is similar to the tangent. The rest of the interval is approximated using polynomials.

Figure 5-13 shows the error of different polynomial approximations:

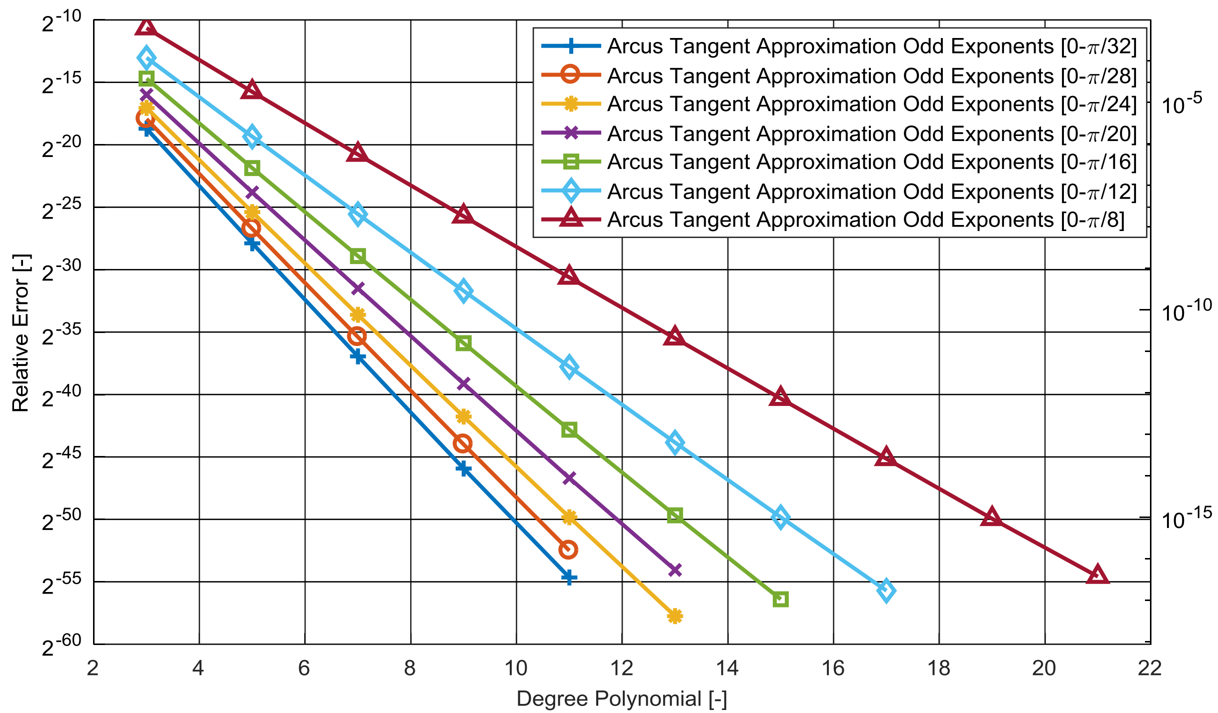

Figure 5-13: Different Arcus Tangent Approximation Polynomials

Similar to the sinus approximations, the polynomials only including odd exponents show almost the precision as polynomials with all exponents with an equal degree. Considering the timing aspect, the odd exponents polynomials will lead to a better performance. Especially considering that the input to the polynomial approximation is $[-X_1, X_1]$ a polynomial containing only odd exponents, needs no special handling of the sign, as it is an odd function like the arcus tangent. Therefore, the following considerations focus only on the variants containing purely odd exponents. Figure 5-13 also shows the expected effect that the polynomials approximating a smaller range reach the same precision with a lower polynomial grade than the ones approximating a bigger range. So in order to determine which variant provides the lowest WCET for a given precision, different variants are implemented and analyzed using aiT.

Proofing just the precision of the polynomials evaluated using double precisions is straightforward. It is done using Gappa and the flow as described in chapter 4.1. The complexity is introduced by the range reduction as shown in the previous chapter.

5.3.5 Algorithm Selection and Implementation Considerations

To reach a high relative precision of 2^{-50}, the following grades were selected:

- Degree 11 for the approximation up to $\tan(\pi/28)$ correlation to $s = 7$: `atan_pi28_11`
- Degree 15 for the approximation up to $\tan(\pi/16)$ correlation to $s = 4$: `atan_pi16_15`
- Degree 17 for the approximation up to $\tan(\pi/12)$ correlation to $s = 3$: `atan_pi12_17`
- Degree 21 for the approximation up to $\tan(\pi/8)$ correlation to $s = 2$: `atan_pi8_21`

In addition to reach a relative precision of at least 2^{-23}, which corresponds to one ulp of a single number, also the following variants were selected:

- Degree 5 for the approximation up to $\tan(\pi/28)$ correlation to $s = 7$: `atan_pi28_5`
- Degree 7 for the approximation up to $\tan(\pi/16)$ correlation to $s = 4$: `atan_pi16_7`
- Degree 7 for the approximation up to $\tan(\pi/12)$ correlation to $s = 3$: `atan_pi12_7`

- Degree 9 for the approximation up to $\tan(\pi/8)$ correlation to $s = 2$: `atan_pi8_9`

In case of the approximation up to $\tan(\pi/8)$ which corresponds to $s = 2$ the interval search is implemented with an “if”, an “else if” and an “else” instruction. The flow chart for the range reduction of this variant is given in Figure 5-14.

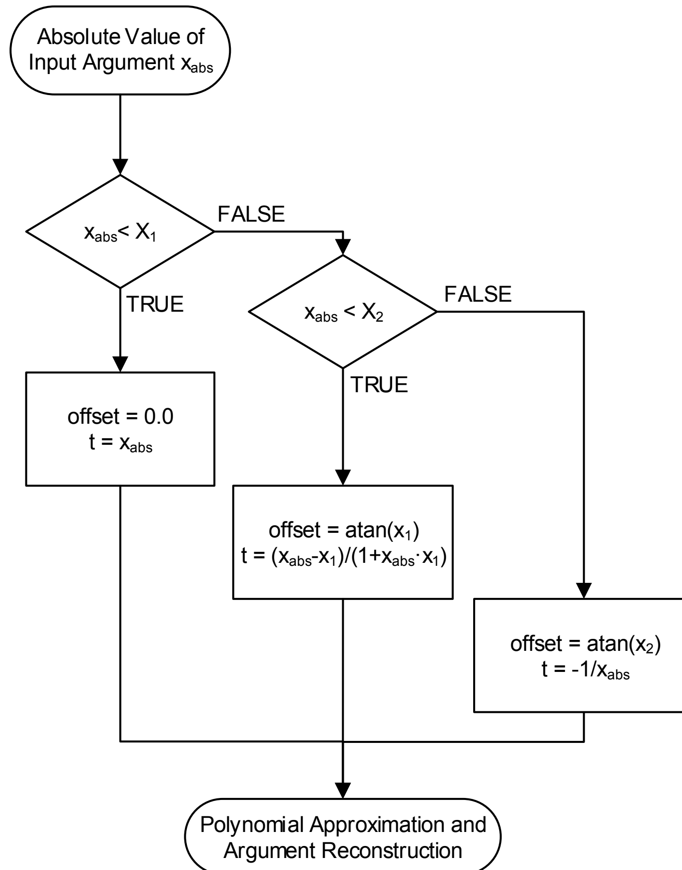

Figure 5-14: Arcus Tangent Range Reduction for s = 2 Variant

Because as only two break points exist a binary search makes no sense in this case. For other variants, a binary search is implemented utilizing nested “if” and “else” conditions. As displayed in Figure 5-15 and Figure 5-16.

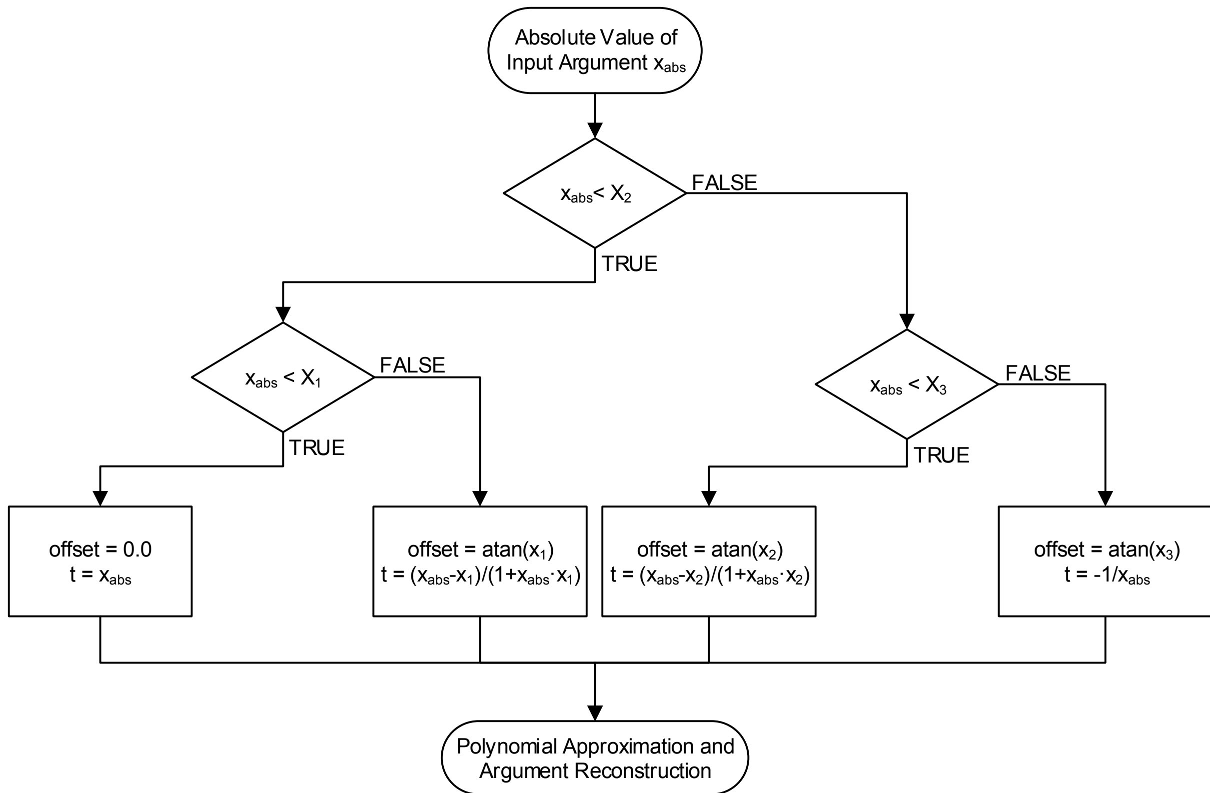

Figure 5-15: Optimal Binary search for the s = 3 Variant

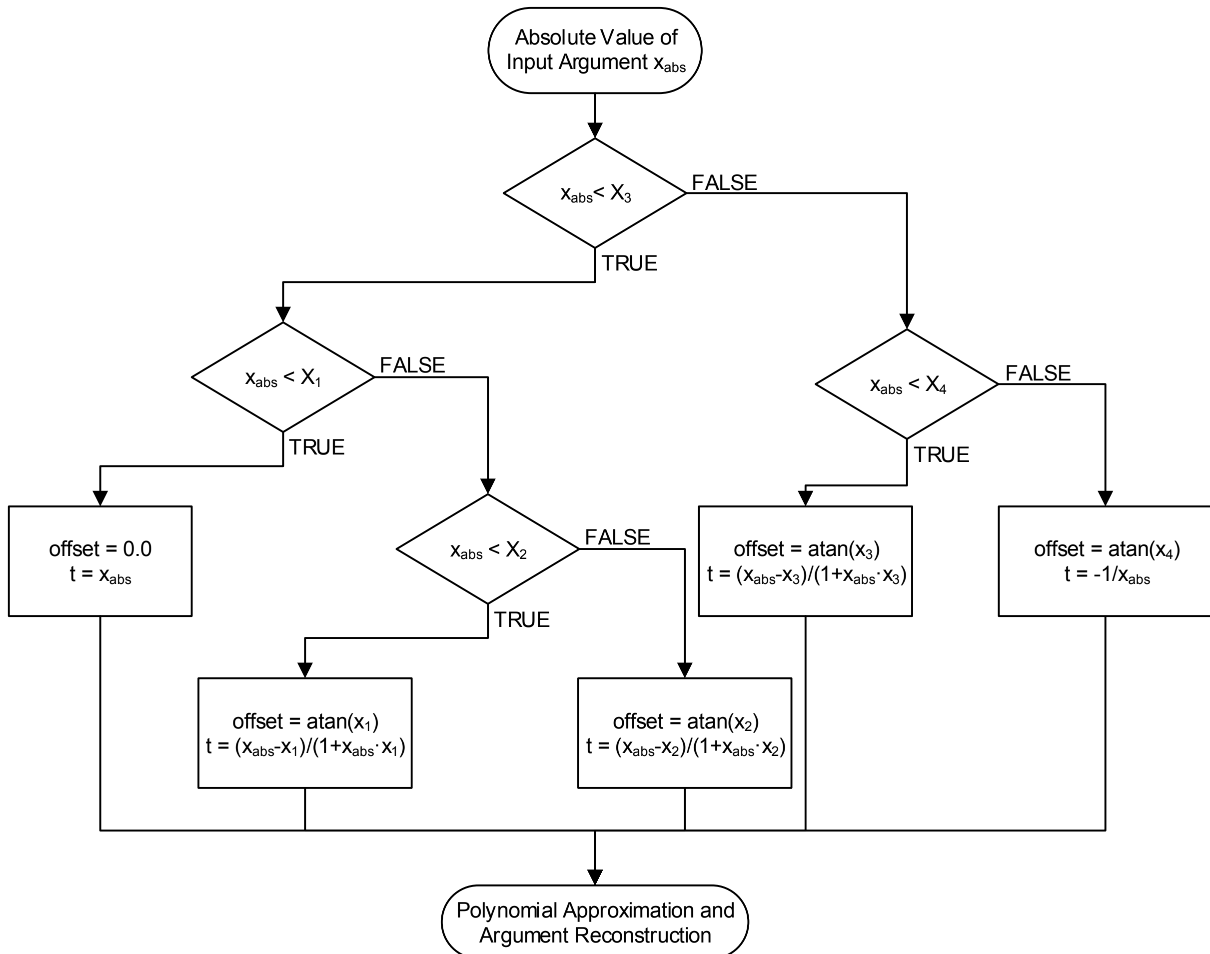

Figure 5-16: None Optimal Binary Search for the s = 4 Variant

An alternative implementation could be done by a while loop, like it is used in the table lookup functions from Simulink. (Compare Listing 8-3) The benefit of a while loop is that it can be used for various numbers of breakpoints. But as here a library function is developed, it is acceptable that the implementation is specific, because this brings a benefit for the WCET analysis of function. In case of the implementation with nested “if else” statements the whole loop is already unrolled in the C source code and so no loop unrolling problems will occur during the WCET analysis. Of special interest for the binary search are the variants with an approximation to $\tan(\pi/12)$ which corresponds to $s = 3$ and $\tan(\pi/28)$ which corresponds to $s = 7$. For those the depths of the binary search is exactly the same for all input variables, and consequently the width of the polynomial approximation is the smallest for a given maximal depth of the tree. This should lead to the best performance regarding precision and WCET. In case that the absolute value of the input to the arcus tangent function is smaller than X_1, the range must not be reduced and the calculation of t can be omitted by an assignment. If the absolute value of the input is bigger than X_s, then due to overflow the first part of equation (5-53) is not valid. In this case t is calculated using the negative reciprocal of the input. For that reason, these two cases are handled in a different way as the other cases. The calculation of the mid cases can be handled using the same instructions. So the calculation can be done in a separate statement after the binary search. This leads to a smaller code size of the function than adding the calculation separate in each case. But as the calculation for the first and the last interval differs, it must be ensured that in those cases the calculation is performed differently. This can either be done by excluding the limit cases from the binary search or by additionally setting a flag during the binary search. Both variants lead to an increased WCET. The second one will introduce problems especially related to the limitation of the abstract interpretation. Both range reductions, the one for the upper limit case contained direct in the corresponding branch and the one after the binary search, will be on the WCET path, although these cases are exclusive in reality. This is due to the merge of value ranges at the end of different branches [57]. This can be either be solved by an annotation in the WCET analysis or by adding the “normal” range reduction into the binary search, in the cases it applies. For a better portability the second variant was chosen. By optimizing the binary search a bit, the increase of the code size can be minimized. In the case of $s = 3$ less than ten additional assembler instructions are necessary, which is acceptable.

Figure 5-17 shows the result of the WCET analysis:

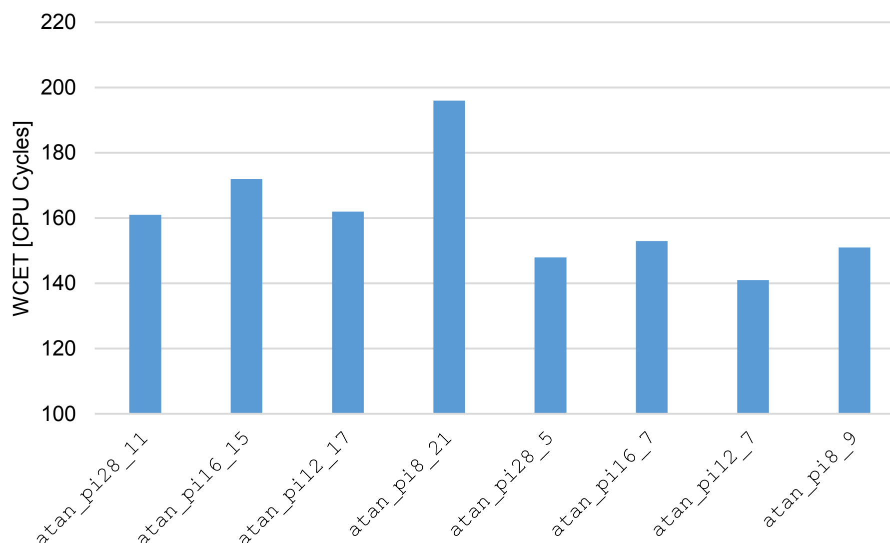

Figure 5-17: Analyzed WCET for Arcus Tangent Function

As expected, the WCET for the variants, which have an optimal binary search tree is lower than the ones the one next to these variants. This can be seen if the `atan_pi12_` (s = 3) variants are compared with the `atan_pi8_` (s = 2) and `atan_pi16_` (s = 4) variants. If a high precision is needed, the `atan_pi12_17` variant is the fastest one although the WCET of the `atan_pi28_11` is only one cycle higher. Considering the fact that the `atan_pi12_17` variant also needs viewer memory both for code and data, it is clear that this variant is selected. In case only the precision of a single number is needed, then the WCET of `atan_pi12_7` variant is clearly the lowest.

The way how to proof the precision of the range reduction is already presented in the previous section. Here the proof of the final reconstruction step and the overall results follow. For the final reconstruction step, the Gappa script is straightforward. The double representative of the offset, and the corresponding relative error is calculated using the high precision of Sollya. For the two intervals in the middle the absolute error of the polynomial approximation, calculated in the previous step, is considered. For the first and last interval the relative error of the polynomial approximation is considered. Altogether the Gappa script for the final reconstruction is: The double precision sum of two approximates for the transcript of the C code, and for the mathematical definition the sum of two values. For each of the four intervals a separate script is compiled. For each script one of the summands is a constant and the other one is variable in a small range.

Table 5-5 gives an overview of the final and intermediate errors for the `atan_pi12_17`:

Range Input Interval	Error Range Reduction	Error Polynomial Approximation	Total Error
$[0.0,0.26]$	N/A	$e_{apr}^{rel} = 2^{-52.2}$	$e_{tot}^{rel} = 2^{-52.2}$
$[0.26,1.0]$	$e^{abs} = 2^{-53.8}$	$e_{apr}^{abs} = 2^{-52.9}$	$e_{tot}^{rel} = 2^{-50.2}$
$[1.0,3.73]$	$e^{abs} = 2^{-52.8}$	$e_{apr}^{abs} = 2^{-52.3}$	$e_{tot}^{rel} = 2^{-51.4}$
$[3.73,\infty]$	$e^{rel} = 2^{-53}$	$e_{apr}^{rel} = 2^{-51.5}$	$e_{tot}^{rel} = 2^{-52.2}$

Table 5-5: Precisions Reached by `atan_pi12_17` on Different Intervals

Next to the arcus tangent functions also the `atan2` functions are provided by the implementation at hand. The approximation of this function is completely based on the `atan_pi12_17` function for the double variant and on `atan_pi12_7` for the single variant. In difference to the arcus tangent function, here two arguments are passed. Based on the sign and the ratio of the input arguments the return value is calculated. Table 5-6 shows the mapping of the arguments input arguments to the calculation where y is the first argument and x the second argument passed to the function.

Condition	Calculation
x > 0	$atan(y/x)$
y > 0	$\pi/2 - atan(x/y)$
y < 0	$-\pi/2 - atan(x/y)$
y = 0 & x < 0	π
y = 0 & x = 0	0

Table 5-6: Conditions for the `atan2` Calculation

The conditions of Table 5-6 are implemented in the same order as shown in the table in nested “if else” statements. For calculation of the arcus tangent the corresponding function for the single or double variant is called. The calculations of the difference always lead to results

where the absolute value is bigger than the absolute value of the inputs. Consequently, no cancelation can occur and the precision considerations of the pure `atan` function keep their validity also for the results here.

5.3.6 Function Stub for the Runtime Error Analysis

The range of the return argument of the arcus tangent is limited to $\pm\pi/2$. Therefore, the easiest version of a stub is a return argument in the range of $\pm\widetilde{\pi/2}$, which is independent from the input to the function. In case of the analyzed software this assumption was sufficient. That the assumption is valid for the chosen implementation is easy to show.

Due to the small relative error, only the last input interval from $[3.73, \infty]$ must be considered, to show that the assumption holds to true. For values bigger than $\tan(5\pi/12)$ the arcus tangent is calculated as: $\widetilde{\pi/2} \oplus \operatorname{atan}(-1 \oslash |x|)$. So it must be shown that $\operatorname{atan}(-1 \oslash |x|)$ is smaller or equal to zero. As the following condition evaluated in double precision is true for all positive non infinite double numbers $|x|$: $-1 \oslash |x| < 0$, it is sufficient to show that the approximation on the reduced range also resolves to a negative value in this case. The two parts of the approximation are considered separately:

First the linear approximation:

With x_{red} being the reduced input argument, for $x_{red} \in]-2^{-26}, 0[$ the approximation is $x_{apr} = x_{red}$ it follows $x_{abs} \in]-2^{-26}, 0[$

Second the polynomial approximation:

In case the relative error of the reduced argument is bound to 2^{-53} the total absolute error of the polynomial approximation is bound by 2^{-53}. As the arcus tangent is strictly increasing, it is sufficient to check the upper bound, in order to show that the polynomial approximation always returns negative values for $x \in [-0.26, -2^{-26}]$. The evaluation of $\operatorname{atan}(-2^{-26}) + 2^{-53}$ with a high precision using Sollya still returns a negative value.

With these results and considering the rounding error of the sum, the assumption $\pi/2 \oplus \operatorname{atan}(-1 \oslash |x|) \leq pi/2$ holds true.

The stub for the `atan2` function is similar as the stub for the `atan` function. In difference to the arcus tangent, the return range is twice as big and set to $\pm\tilde{\pi}$. In order to show the validity of this assumption, the calculation of the `atan2` must be considered. For return values in the second and third quadrant the function value is calculated as the difference of $\pm\widetilde{\pi/2}$ and an arcus tangent value. As shown above the arcus tangent always returns values in the range of $\pm\widetilde{\pi/2}$. To proof the validity the following must be shown:

$$\widetilde{\pi/2} \ominus \widetilde{-\pi/2} \leq \tilde{\pi} \tag{5-60}$$

$$\widetilde{-\pi/2} \ominus \widetilde{\pi/2} \geq -\tilde{\pi} \tag{5-61}$$

The number $\widetilde{\pi/2}$ is exactly the half of $\tilde{\pi}$ and for both numbers the three LSBs of the significand are zero. As the sign does not introduce an uncertainty, instead of the two equations also just the sum of $\widetilde{\pi/2}$ with itself can be considered. As long as no overflow occurs, the sum of a binary floating-point number with itself is always exact. Consequently, the calculations in the equations (5-60) and (5-61) are also exact and the return value of the stub is valid.

The ANSI C does give the possibility that an error is raised for the case that both inputs to the `atan2` are zero [47]. For that reason, the stub of the `atan2` is also extended by a check for both arguments being zero. In case the check evaluates to true an alarm is raised in the analysis.

5.4 Arcsine and Arccosine

The arcus sine and arcus cosine are the inverse function of the sine and the cosine. The valid input range for both functions is $[-1.0,1.0]$. The range of the return argument is $[-\pi/2, \pi/2]$ for the arcus sine and $[0, \pi]$ for the arcus cosine. For both functions the derivative tends to infinity at the limits of the defined range. Similar to the sine and cosine both functions can be transferred in each other by a simple sum. To calculate the arcus cosine the following equation can be used:

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x) \tag{5-62}$$

The application software at hand has 28 call sites to the arcus cosine and the arcus sine function. As shown in Table 3-1, for the arcus sine the majority of the call sites calls the single variant of the function. In contrast to this, for the arcus cosine only the double variant is use.

5.4.1 Input Restrictions

As shown before both the arcus sine and the arcus cosine are limited to an input range of $[-1.0,1.0]$. If the functions are used in the SLCI compatible Simulink environment, the Embedded Coder produces some protection statements before the actual library function is called. This protection code limits inputs, which are outside of the boundary of ± 1.0, to the corresponding limit value. In order to ensure the same behavior in case the function is integrated as legacy code, the implementation also shall return the same value, which is returned for ± 1.0 for inputs outside of the defined range. In case of small numerical errors this leads to a stable behavior. This behavior is in-line with the implicit behavior of Simulink but it is contradicting to the ANSI C, which defined that a domain error shall occur in such cases [47]. Similar to the ANSI C the value should be returned in radians.

5.4.2 Architecture

The fact that the derivative of the function tends to infinity at the bounds leads to problems when the arcsine shall be approximated by a polynomial. To solve this, different solutions exist. One is proposed in [11], here it is suggested to use the following equation which is based on the arcus tangent and the square root:

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right) \tag{5-63}$$

As this equation is based on two functions, the arcus tangent and the square root which also need to be approximated by a routine, the computational effort for such an approach might be huge. A different approach is used in [27], here the positive input range $[0,1.0]$ is divided into ten subdomains. For arguments in the first one close to zero, a low degree polynomial is calculated. For the eight intervals in the middle, a polynomial of the same degree is used. The same degree enables the reuse of the code calculating the polynomial. But this brings the downside that in case the precision on each interval shall be of the same magnitude, the length of the intervals varies. So in order to determine the right interval, a search is necessary. For

values within the range of the last interval, a polynomial p is combined with a square root and an offset [27]:

$$\arcsin(x) \approx p(1-x) * \sqrt{2-2*x} + \frac{\pi}{2} \tag{5-64}$$

Due to the offset and the factor the polynomial can approximate a slightly curved function, instead of the arcus sine which is heavily curved in this section. The offset in equation (5-64) is equal to the return value of the arcus sine with an input value of 1.0:

$$\arcsin(1.0) = \frac{\pi}{2} \tag{5-65}$$

The factor is similar to the inverse derivative of the function:

$$arcsin'(x) = \frac{1}{\sqrt{1-x^2}} \tag{5-66}$$

But instead of evaluation the radicand with the squared x a Taylor series of $1-x^2$ is used. The Taylor series starting at 1.0 considering only the first two terms resolves to:

$$(1-1^2) + (-2*1)*(x-1) = 0 - 2*(x-1) = 2-2x \tag{5-67}$$

As both variants require that a square root is calculated, it is difficult to predict which variant provides the lower WCET. To find this out, both variants are examined. For both variants the implementation first takes the absolute value according to the code presented in section 4.2. Based on the flag and the property that the arcus sine is an odd function, the return argument is restored in the last step. Therefore, in the following without loss of generality only positive values are considered.

5.4.3 Range Reduction

The input range of the inputs to the arcus sine and arcus cosine functions are quite limited. But for the version which is based on the proposal of [27] still a range reduction is necessary. But as stated above the bounds are determined based on the polynomial evaluation and do not follow a certain pattern. So, in this case the range reduction is implemented by a binary search checking for the bounds determined during the development of the core algorithm.

For the version which is based on the arcus tangent no special range reduction is applicable.

5.4.4 Core Algorithm

Direct Implementation

In this section the direct implementation according to the implementation in [27] is described. In difference to the work there the designed function does not need to be correctly rounded. For that reason, only the following points are transferred:

- Use polynomial approximation with the same order in the mid part.
- Instead of a direct approximation of the function in the last interval use the factor $\sqrt{2-2*x}$ and the offset $\pi/2$.

The grade of the used polynomials is different to the implementation in [27].

The implementation at hand uses a linear approximation for input values close to zero. Similar to the tangent the range in which such an approximation is valid Sollya is used. The

findzeros command is applied. The results for the range of the linear approximation are given in Table 5-7 where the upper bound is rounded down to 2^{-n}, with n beeing an integer number.

Desired Relative Error	Range
$e^{rel} \leq 2^{-53}$	$[0.0, 2^{-26}]$
$e^{rel} \leq 2^{-51}$	$[0.0, 2^{-25}]$
$e^{rel} \leq 2^{-50}$	$[0.0, 2^{-24}]$

Table 5-7: Range of Linear Approximation of the Arcus Sine

Therefore, in order to get a precision of 2^{-51}, which is similar to the precision reached by the arcus tangent, 2^{-25} is selected as upper bound for the linear approximation.

The length of the midpoint intervals is determined by the precision which shall be reached and by the grade of the polynomial. This problem cannot be solved analytically. Thus this problem is solved by an iterative Sollya script. As the precision, which shall be reached, is normally known a prior, the three dimensional problem can be reduced to a two dimensional problem, by setting the desired precision. The selection of the degree is a discrete problem, so it is easy to iterate over different degrees. In order to determine the ranges on which the approximation reaches the desired precision the upper bound of the approximation is decreased. In order to approximate the whole desired range, normally more than one different polynomial is calculated. A detailed flow graph for this calculation is presented in Figure 5-18.

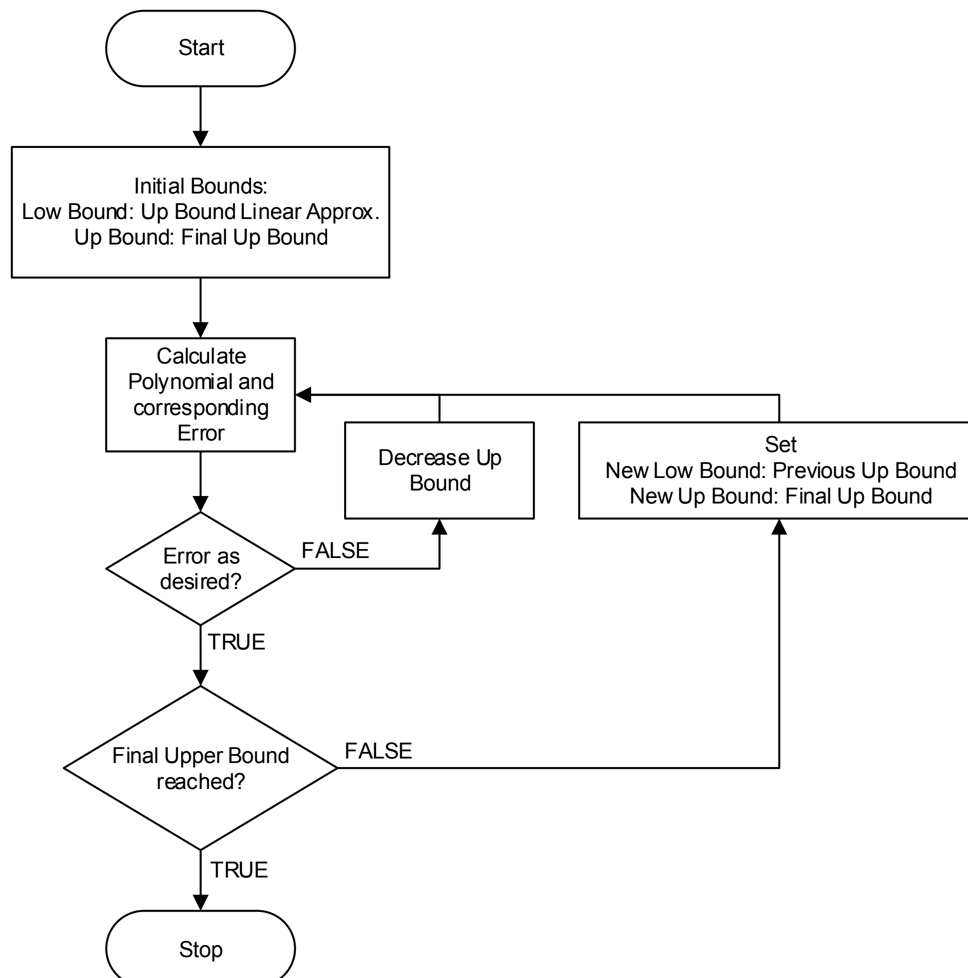

Figure 5-18: Flow Graph of Sollya Script to Determine Ranges and Number of Approximations for the Arcsine

Additionally to the loops shown in the figure, one outer loop is added in the script to vary the degree of the polynomial. The desired final up bound shown in the flow graph is dependent on the validity of the approximation in the last interval. To find a good tradeoff in the execution time for calculation the middle parts and the last interval, the final up bound was manipulated manually. As stated before for the range reduction, a binary search is implemented in order to determine the applicable coefficients of the polynomial. Therefore, the variants which need two, four or eight different polynomials are of special interest, because in those cases the depth of the binary tree to all elements is constant. Two variants of polynomials are considered. The first variant is approximating the arcus sine minus the arcus sine of the low bound l_i of the interval.

$$\arcsin(x) \approx \arcsin(l_i) + p_i(x - l_i) \qquad (5\text{-}68)$$

And the second one which is similar to the approximation in the "crlibm" [27]. Here a polynomial approximates the arcus sine subtracted by the arcus sine of median m_i of the interval.

$$\arcsin(x) \approx \arcsin(m_i) + (x - m_i) \times p_i(x - m_i) \qquad (5\text{-}69)$$

Both polynomial approximations are calculated for a reduced range. The first variant subtracts the low bound of the range from the input, which leads to a range for the approximation from zero to the width of the interval. So in case the reduced argument is zero, in the ideal case the polynomial is also zero. This leads to difficulties by calculating the approximation as the relative error for zero is not defined. The problem can be omitted if the low bound of the reduced range is increased to the next possible double number and the exact zero case is afterwards considered manually. The second variant has a reduced range from minus half interval width to plus half interval width. Here the polynomial is calculated for an absolute error in order to omit the zero case problem. For the selected overall range $[2^{-26}, 0.854]$ Table 5-8 shows the necessary degrees of the polynomial.

Offset	8 Intervals	4 Intervals
Low Bound	11	15
Median	11	15

Table 5-8: Polynomial Grades for Arcus Sine Approximation. Range: $[2^{-26}, 0.854]$

As the degree of the both variants is the same, the one with the offset based on the low bound is selected. The reason therefore is the expectation, that the proofs to validate the reached precision during an execution with double precision are easier when the zero case is at the limit of the interval.

For input values close to one the approximation is calculated as in equation (5-64). That means the square root needs to be calculated. As this calculation has a huge computational effort, the polynomial in this case should have a low degree. As for the midpoint sections, different degrees were examined for the polynomial approximation in the last interval. For a given degree, the value of the low bound of the approximation is iteratively adjusted until the desired precision is reached. With lower degrees the desired precision is only achieved on a smaller interval. This can lead to problems as the polynomials in the middle section do not have a good convergence for big values. Consequently, a tradeoff must be determined here. This is done by executing the algorithm for the middle section polynomial with different values for the necessary upper bound. As a good tradeoff, the variant with a degree of seven was selected. In case of the desired precision of 2^{-51} this approximation is valid for input values in the range of $[0.854, 1.0]$.

In order to show the precision reached by the algorithm evaluated in double precision on the target, Gappa scrips are compiled. For the intervals with the direct approximation these are straightforward according to section 4.1. For the last approximation where the square root is part of the approximation, the precision of the implemented square root function has to be considered. First the precision of the radicand has to be considered. Instead of $2 - 2 * x$ in equation (5-64) the implementation calculates $2 * (1 - x)$. In this case $1 > x \geq 0.5$ so Sterbenz lemma can be used to show that no rounding error occurs in the first step of the calculation of the radicand. As the double numbers are binary based, the multiplication with two is also without error. Therefore, only the error of the square root function must be considered. This is done by introducing in the mathematical definition part a variable assigned to the exact square root. In the transcript of the code, the approximation of the square root is only used in the last instruction where the actual arcus sine is calculated. The definition is through a given relative error of the square root function in the theorem section. This error is set to $\leq 2^{-52}$ which correlates to the error of the square root function without a final rounding test (Compare chapter 6). In order to get an error, which is reached by the approximation polynomial the `supnorm` command is not applied to the calculated polynomial. Instead, the following command is used, where `fp` is the polynomial returned by the `fpminimax` approximation function:

```
1  supnorm(0,fp(1-x)*sqrt(2-2*x)+a-asin(x),[0.85…,1-2^-53],absolute,1b-10);
```

Listing 5-6: Sollya Code to get Absolute Error of the Arcus Sine Approximation in the last Interval

In the listing above `a` is equal to the double precision representative of $\pi/2$ and `fp` is the result of the approximated polynomial.

This command calculates the absolute error of the approximation. The reason for not setting the polynomial to `fp` is that the `sqrt(2-2*x)` would be part of the denominator of the approximated function which leads to numeric instability for values of `x` close to 1.0. It is also not possible to consider the square root in the first argument of the `supnorm` function because there only polynomials are accepted [28].

The results for all intervals are given in Table 5-9:

Range Input Interval	Total Relative Error
$[0.0,0.38]$	$e_{tot}^{rel} = 2^{-50.9}$
$[0.38,0.63]$	$e_{tot}^{rel} = 2^{-50.9}$
$[0.63,0.79]$	$e_{tot}^{rel} = 2^{-51.5}$
$[0.79,0.87]$	$e_{tot}^{rel} = 2^{-52.1}$
$[0.87,1.0]$	$e_{tot}^{rel} = 2^{-50.7}$

Table 5-9: Precisions Reached by Direct Arcus Sine Implementation

Derived from the arcus tangent

Equation (5-63) has a singularity for input values equal to 1.0. Consequently, this case is handled in an extra statement. The return value is set to the double representative of $\pi/2$ in the case that the input argument is bigger or equal than 1.0. The reason for also including arguments which are bigger than 1.0 is to implement the same behavior as the normal arcus sine implementation of Simulink. Also values smaller than 2^{-26} are directly approximated by a linear approximation. This also eases the proofs for the reached precision.

For all other cases the equation (5-63) is used. The radicand can be calculated using a FMA instruction. The denominator consists of a square root. To omit an expensive division, the square root function is extended such that the reciprocal square root is returned, as proposed in chapter 6. Therefore, the division can be omitted by a faster multiplication. After multiplying the input argument with the reciprocal square root, the arcus tangent function is called to calculate the return argument.

In order to proof the precision reached by the algorithm executed on the target, Gappa scripts are compiled. In difference to the direct implementation here a combination of different scripts is necessary. Mostly due to the fact that the implementation is a combination of different functions, but also due to the fact that the arcus tangent proof already consists of more than one script. The main Gappa script contains the transcript of the C source code of the corresponding function. Also, the mathematical definition for all statements are contained in the script. For proofing the precision of the approximation of the reciprocal square root, the separate square root script is used. Analogous to the last interval of the direct implementation, just the mathematical definition of the reciprocal square root is contained. In the transcript section only a variable is introduced. This variable is defined via a relative error of the mathematical correct companion. In comparison to the script used for the pure square root, the difference is, that the input to the square function already has a relative error. This error can be deducted from the calculation of the radicand. The relative error of the reciprocal square root calculation is then transferred to the main script in order to proof the precision of the arcus sine. With the error of the square root also the error of the multiplication can be calculated before the intermediate result is passed to the arcus tangent. Additionally, the relative error of the reduced argument for the arcus tangent is calculated in the case the reduction is $-1/x$. This is done because therefore no script in the arcus tangent case exists. There the range reduction is trivial, as it is only a correctly rounded instruction with a non-rounded argument as input. Afterwards, the same scripts as for the tangent are executed but considering the calculated relative error of the precedent instruction. Table 5-10 shows the relative error of the tangent approximation in this cases, as the tangent is the last instruction with rounding the error of the arcus sine is the same.

Range Input Interval	Error Range Reduction	Error Polynomial Approximation	Total Error
$[0.0,0.26]$	N/A	$e_{apr}^{rel} = 2^{-50.7}$	$e_{tot}^{rel} = 2^{-50.7}$
$[0.26,1.0]$	$e^{abs} = 2^{-53.0}$	$e_{apr}^{abs} = 2^{-52.4}$	$e_{tot}^{rel} = 2^{-50.0}$
$[1.0,3.73]$	$e^{abs} = 2^{-51.5}$	$e_{apr}^{abs} = 2^{-51.3}$	$e_{tot}^{rel} = 2^{-50.5}$
$[3.73,\infty]$	$e^{rel} = 2^{-51}$	$e_{apr}^{rel} = 2^{-50.4}$	$e_{tot}^{rel} = 2^{-51.7}$

Table 5-10: Precisions Reached by the Arcus Sine Implementation via the Arcus Tangent

In comparison to the direct implementation, the memory effort of this implementation is quite low as no separate coefficients are necessary. But as the function is built on two other complex functions, the calculated WCET is with 438 cycles significantly higher than the one for the direct implementation.

5.4.5 Arcus Cosine

As stated before, to calculate the cosine the following equation can be used:

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x) \tag{5-70}$$

When equation (5-70) is executed using double precision arithmetic, the result can be subjected to catastrophic cancelation for values of x close to 1.0., as the arcus sine is in the same magnitude as $\pi/2$ and both are approximations of real numbers. Therefore, if equation (5-70) is directly implemented, multi precision arithmetic would be necessary. This can be omitted by considering how the arcus sine is approximated in the previous section. For values close to one, equation (5-64) is used. The polynomial approximation is multiplied by $\sqrt{2-2x}$ and $\pi/2$ is added. In case the arcus cosine for values close to one is calculated, not the approach of simply subtract the arcus sine of $\pi/2$ shall be used. Instead the consecutive addition and subtraction of $\pi/2$ is omitted, by directly approximating the result with the negated polynomial p of the arcus sine times the square root.

$$\arccos(x) = -p(1-x) * \sqrt{2-2*x} \tag{5-71}$$

As the range of the return value is different between the arcus sine and the arcus cosine, the results of the precision proofs are not transferable without adaption. First, the case that the input argument is bigger than 0.87, is considered. For the proof of the arcus sine, it was sufficient to work with an absolute approximation error. As the return value of the arcus sine is greater than 1.0, in this interval the relative approximation error is smaller than the absolute error. For the arcus cosine, the absolute error is not sufficient to proof a tight bound of the relative error in this interval. The reason is that the range of return value of the arcus cosine is $[0.0,0.52]$ for an input in the range $[0.87,1.0]$. Thus, the exponent of the return value has a greater range than for the arcus sine case. As the return value is smaller than 1.0, also the assumption that the absolute and the relative error are equal is not valid. As the input value of 1.0 is handled in a separate statement, the upper bound of the input range can be restricted to the double precision number closest to one which is $1-2^{-53}$. The corresponding value of the arcus cosine is:

$$\arccos(1-2^{-53}) \approx 1.5 * 10^{-8} > 2^{-26} \tag{5-72}$$

Consequently, subnormal numbers must not be considered in the calculation. But still with this restricted range it is not possible to determine a relative error of the approximation using Sollya. In order to omit this problem, the relative errors are calculated via the absolute error on subintervals. This is done using a Sollya script. The Control flow of this script is shown in Figure 5-19:

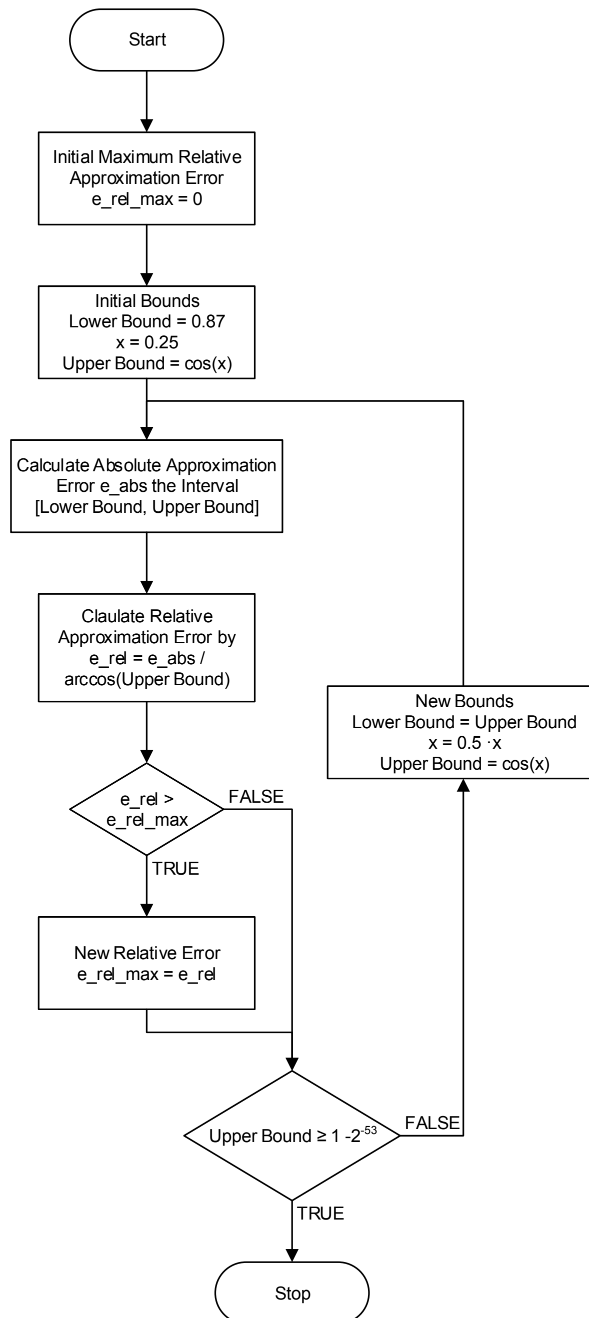

Figure 5-19: Sollya Script for the Calculation of the Arcus Cosine Relative Approximation Error

The script starts with the interval $[0.87, \cos(0.25)]$. As the $\arccos(0.87)$ is a bit larger than the 0.5, it is ensured that the exponent of the results is almost the same. Therefore, the relative error can be approximated with an acceptable bound by dividing the absolute error through the smallest return value of the function. As the arcus cosine is monotonic decreasing, the smallest return value can be determined with the biggest input value. In the next step, the lower bound of the input is set to the upper bound of the previous step. For the calculation of the upper bound the argument of the cosine is halved. Thus, it is ensured that the range of the exponent is small for all subintervals. This is repeated until the whole range is covered. For all intervals, it is checked if the present relative error is bigger than the previous maximum. In case this is true, the maximum is overwritten by the present value. So, at the end a good approximation of the maximum of the relative approximation error for the whole interval is gained. The statement for the calculation of the approximation error is similar to the arcus sine, presented in the previous section. The addition of the double representative is not applied and instead of the

subtraction of the arcus sine the arcus cosine is added. With this approach, the relative approximation error can be bound to approximately $3.4 + 10^{-16}$. This error is applied in the Gappa script. The script works similar to the arcus sine and a total relative error of $2^{-50.2} = 7.9 * 10^{-16}$ can be proven for the last interval.

For the rest of the input interval equation (5-70) is directly implemented to calculate the arcus cosine. For negative input values the arcus sine becomes also negative. Therefore, the result will be bigger than the input values and no cancellation will occur. For the positive range $[0.0, 0.87]$, the maximum value of the arcus sine is approximately 0.51. Therefore, here no complete cancelation can occur. In order to show the precision actually reached by the implementation, a Gappa script for this part of the implementation is compiled. The script is straightforward, because it basically only contains the difference of two approximated symbols, and their mathematical definition. For both approximations the relative error is given, for $\widetilde{\pi/2}$ the error is calculated with Sollya and for the error of the arcus sine the result of the proofs for this function are applied. Additionally, also the range of the arcus sine and the arcus cosine is given. The relative error of the arcus cosine function is $e^{rel} = 2^{-49.8} = 9.9 * 10^{-16}$.

5.4.6 Algorithm Selection and Implementation Considerations

Figure 5-20 shows the WCETs for the arcus sine variants:

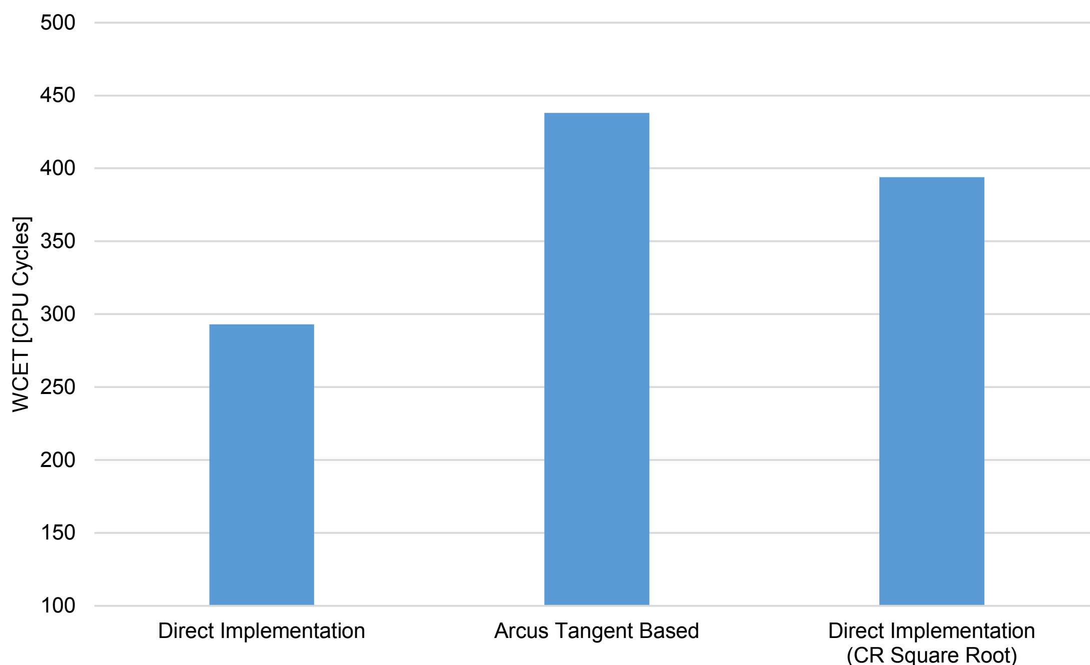

Figure 5-20: Analyzed WCET for Arcus Sine Function

The direct implementation is clearly faster than the variant based on the arcus tangent. Consequently, in case that there are no memory restrictions, the direct implementation should be preferred. The limitation of the main memory is not a problem for the system at hand, but as cache locking is applied during the integration, the memory consumption of the library function still might be a topic. As shown later in chapter 9, the memory footprint of the libraries is bigger than one cache way. But in case more ways are locked, the overall memory consumption of the libraries is not a problem. For that reason, for the system at hand the direct implementation method is implemented. In addition to the variants discussed in this subsection the WCET for the direct implementation with a correctly rounded (CR) square root is shown.

Here the execution time benefit is smaller, but as the arcus tangent based implementation utilizes the not correctly rounded reciprocal square root, this is an unfair comparison. But for integration reasons the square root might perhaps only be available in the correctly rounded version. Therefore, the WCET of this variant might also be of interest. As displayed in the diagram this variant still has an execution time benefit over the arcus tangent based variant.

A further benefit of the direct implementation method is that the arcus cosine calculation can be based on equation (5-70). The potential cancelation for input values close to 1.0 can be omitted here as shown in the previous section. In case of the variant based on the arcus tangent, this approach is not feasible. To omit the cancelation in this case, a different approach must be applied. A potential solution can be deviated from the half angle equation. As start the following equation is used [88]:

$$\tan^2\left(\frac{z}{2}\right) = \frac{1-\cos(z)}{1+\cos(z)} \tag{5-73}$$

By setting $z = \arccos(x)$ equation (5-73) resolves to:

$$\tan^2\left(\frac{\arccos(x)}{2}\right) = \frac{1-\cos(\arccos(x))}{1+\cos(\arccos(x))} = \frac{1-x}{1+x}\ with - 1 < x \leq 1 \tag{5-74}$$

It can be further simplified by taking the square root on both sides:

$$\tan\left(\frac{\arccos(x)}{2}\right) = \sqrt{\frac{1-x}{1+x}}\ with - 1 < x \leq 1 \tag{5-75}$$

Solving this equation for $arcos(x)$, leads to:

$$\arccos(x) = 2\arctan\left(\sqrt{\frac{1-x}{1+x}}\right)\ with - 1 < x \leq 1 \tag{5-76}$$

So, in case of an implementation based on the arcus tangent the arcus cosine can be based on this equation as here no cancelation will occur. Only the limit case of minus one must be handled separately.

The implementation of a dedicated version for single precision input arguments was skipped. The reason for that is that one of the major drivers of the WCET is the call to the square root function. As shown in chapter 6, a single variant of the square root is not implemented, as the WCET would not differ much. As a consequence, also the WCET of a single variant of the arcus sine and arcus cosine would not be much smaller than the double variant. Table 3-1 shows the single arcus cosine is not used at all and the single variant of the arcus sine has twenty call sites. These twenty call sites are not called recursively. Taking all these circumstances into account, the benefit of a potential single variant of the arcus sine on the overall WCET is insignificant.

5.4.7 Function Stub for the Runtime Error Analysis

A simple stub for the arcus sine is that the range of the return argument is defined to $[-\pi/2, \pi/2]$, independent from the input argument. Similarly for the arcus cosine a range of the return value is set to $[0, \pi]$. For the flight control software at hand these assumptions showed that they were sufficient enough that no false alarms were triggered due to a too huge

abstraction of the functional behavior. As stated in the previous section, the direct implementation was selected for implementation.

The validity of the arcus sine stub can be shown with the following consideration: The arcus sine is a monolithically increasing function. As the stub is just a return range independent from the input value, it is sufficient if the limits are considered. For negative input values, the implementation only considers the absolute value and returns the approximation multiplied my minus one. As result, the considerations can be further reduced to the upper input limit. The input value of 1.0 is anyway handled in an extra branch, in which the return value is set to $\widetilde{\pi/2}$. For that reason, the limit value is trivial. In order to show the validity, also for the actual approximation, the arcus sine for the double precision predecessor of 1.0, which is $1.0 - 2^{-53}$, is calculated with a high precision using Sollya. The result is then multiplied by one plus the relative error and compared to the double representative of $\pi/2$.

$$\arcsin(1.0 - 2^{-53}) * \left(1 + e_{tot}^{rel}\right) < \widetilde{\pi/2} \tag{5-77}$$

The result of Sollya shows that inequation (5-77) is valid and the stub of the arcus sine is valid, too.

For the arcus cosine, both bounds must be considered separately. In case of the negative value, the return value is calculated according to equation (5-70). As shown above, the arcus sine is never smaller than $\widetilde{-\pi/2}$. And with the argumentation similar to the arcus tangent, it can be shown that difference of $\widetilde{\pi/2} \ominus \widetilde{-\pi/2}$ never gets bigger than $\tilde{\pi}$. For the positive bound the limit value of 1.0 is handled in a separate branch, in which the return value is set to 0.0. For the predecessor double precision floating-point number of 1.0, it is already possible to show a relative error much smaller than 1.0. Consequently, the approximated value must be bigger than 0.0. Taking these two points into consideration, it is clear that the return value of the arcus cosine implementation is always greater or equal to 0.0. So, all together, stubbing the return value of the arcus cosine with the range $[0.0, \tilde{\pi}]$ is valid.

6 Square Root

The square root is the only function implemented in this thesis, which is requested by the IEEE 754 [3] to be correctly rounded. Although the function is requested by this standard, the hardware at hand does not provide an implementation of this function [15]. This is why here a software implementation of this functionality is necessary.

It is difficult to find proper fitting polynomials for the approximation of the square root [5]. This is partly related to the infinite derivative at zero. For that reason, the Newton-Rapson method is used to calculate the square root. The algorithm in the next section is based completely on the proposal of [49]. The contributions are the addition of a formal proof and the considerations if a correctly rounded version is useful in the given use case.

6.1 Function Implementation

The chosen Newton-Rapson approach does not work for input arguments, which are exactly zero. To omit errors in this case, in the implementation the input argument is checked for being zero before the Newton-Rapson calculation is started. In case the input value is equal to zero, directly the value zero is returned and the calculation of the square root is skipped. The runtime error analysis showed that due to rounding errors, the square root function might be called with arguments close to zero and a negative sign. In order to omit an error in such cases, the check for zero has been extended to check for smaller or equal to zero. This adaptation leads to a robust behavior of the overall software.

In order to calculate the square root of the input a the positive root of the following function can be calculated using the Newton-Rapson approach:

$$g(x) = x^2 - a \tag{6-1}$$

In order to execute the Newton-Rapson method the derivate of the function is needed:

$$g'(x) = 2x \tag{6-2}$$

With equations (6-1) and (6-2) the iteration step resolves to:

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{6-3}$$

This step involves a division by the current estimate x_n. As presented in section 4.2, the floating-point division is an instruction with a high timing penalty and it shall be omitted if possible. In case of the square root approximation with the Newton-Rapson approach, this is possible by a reformulation of the approximated function. Instead of $g(x)$, the following function could be approximated:

$$f(x) = \frac{1}{x^2} - a \tag{6-4}$$

The root of this function is $1/\sqrt{a}$, by multiplying this root with a, $\sqrt{a}$ is the result. So, by setting a equal to the input of the square root function, the desired square root can be calculated.

The derivate of (6-4) is:

$$f'(x) = -\frac{2}{x^3} \tag{6-5}$$

Using the equation (6-4) and (6-5) the Newton-Rapson iteration evaluates to:

$$x_{n+1} = \frac{x_n}{2}(3 - ax_n^2) \tag{6-6}$$

For the start of the Newton-Rapson approach, an initial approximation for the reciprocal square root is needed. For the given target architecture there is a specific instruction giving an approximation of $1/\sqrt{x}$ with a precision of $\frac{1}{32} = 1b - 5$ relative to the correct result [15, 89]. In the implementation at hand, this instruction is used because the latency of this instruction is just three cycles. As it is target specific the instruction is contained in the hardware abstraction layer, as introduced in section 4.2.

If no instruction for the calculation of a reciprocal square root estimate is available, the start value of the Newton-Rapson method can also be calculated by software. The basic idea goes back to the paper draft of W. Kahan and K.C. Ng in the year 1986, which for example can be found in the e_sqrt.c file of [10]. The basic idea behind the algorithm is that an IEEE binary floating-point number can be written as:

$$sign * 2^e * (1 + m) \tag{6-7}$$

Where e represents the unbiased exponent an m the significand without the hidden bit. In case of the square root the sign is positive, which correlates to the $sign$ in equation (6-7) being equal to 1.0. Therefore, it can be neglected and in order to ease the reading in the following, the sign is not written. In case the square root is written as an exponent of 0.5, the square root of a floating-point number resolves to:

$$\sqrt{x} = 2^{e*0.5} * (1 + m)^{0.5} \tag{6-8}$$

Applying a Taylor series approximation to the significand term equation (6-8) resolves to:

$$\sqrt{x} \approx 2^{e*0.5} * \left(1 + \frac{m}{2}\right) \tag{6-9}$$

So, in order to get an estimate for the square root of a floating-point number, both the significand and the exponent must approximately be halved. This can be achieved by interpreting the memory as an unsigned integer number and execute a shift of one to the right. As this is just an estimate, it is sufficient to apply this just to the first 32 bits in a big endian system. As the desired final result is an estimate for the reciprocal square root and not of the square root, equation (6-9) must be inverted:

$$\frac{1}{\sqrt{x}} \approx 2^{e*0.5*-1} * \left(1 + \frac{m}{2}\right)^{-1} \tag{6-10}$$

If the significand term is again approximated by a Taylor series, equation (6-10) can be written as:

$$\frac{1}{\sqrt{x}} \approx 2^{e*0.5*-1} * \left(1 - \frac{m}{2}\right) \tag{6-11}$$

The equation can easily be implemented by shifting and subtraction operations. So, in order to enable these operations, first the high word of the double precision floating-point number is interpreted as an unsigned integer word. Then as stated above, the division by two can be implemented by a right shift. In order to implement the negation of the mantissa and the exponent, the right shifted result is subtracted from the number 0x5fe80000.

```
1  k = 0x5fe80000 - (x0 >> 1);
```

Listing 6-1: First Step of the Reciprocal Square Root Estimate Calculation

The variable k is already a rough estimate for the reciprocal square root $rsqrte$, but the precision is further increased as described afterwards. To demonstrate the meaning of number 0x5fe80000, Table 6-1 displays the mapping to the components of a double precision number. As the variable x0 contains only the 32 most significant bits, the table is also restricted to these, beginning with the MSB on the left hand side. In the first line the bits are given and in the second line the numbers allocated to each component.

0	1	0	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0x5FE = 1534											0x80000 = 524288																			
Sign	Exponent											Significand																			

Table 6-1: Mapping of 0x5fe80000 to the Sig, Exponent and Significant

The table shows the exponent contains the number 1534. For the significand the most significant bit is set.

As equation (6-11) is based on the unbiased exponent but the number is stored with the biased exponent, this factor is necessary. As the exponent is stored with an offset of 1023, for the negation of the exponent the following equation applies:

$$o - (1023 + e) = 1023 - e \tag{6-12}$$

Equation (6-12) resolves to o equal to two times the bias. Prior to the calculation of the difference, in Listing 6-1 the shift is executed, so also the biased exponent is halved. Therefore, instead of twice the bias only 1.5 times the bias is used. This also leads to the fact that the highest bit belonging to the mantissa is set in the constant.

In order to increase the precision of the estimate, W. Kahan and K.C. Ng propose a second step. Based on the six most significant bits of the significand, an offset is loaded out of an array and subtracted from the original estimate. Utilizing this method, a precision of 7.8 bits could be reached. Also, different methods exist where the offset o in equation (6-12) is modified. With this approach precisions up to 4 bits can be achieved.

To take benefit of the fused multiply add instruction, equation (6-6) is reformulated according to [49].

With the definition of a “residual” $\epsilon_n = 1 - ax_n^2$, equation (6-6) resolves to:

$$x_{n+1} = x_n + \frac{1}{2}\epsilon_n x_N \tag{6-13}$$

With the definition of a new variable $r_n = \frac{1}{2} * \epsilon_n$ and by multiplying equation (6-13) with a the equation resolves to:

$$sqrt_{n+1} = sqrt_n + r_n sqrt_n \tag{6-14}$$

This expression can be calculated using a FMA instruction. The calculation of $r_n$is as follows:

$$r_n = \frac{1}{2} - \frac{1}{2} a x_n^2 = \frac{1}{2} - sqrt_n * \frac{1}{2} x_n \tag{6-15}$$

If $\frac{1}{2} x_n$ is defined as h_n the equation above can also be implemented using a FMA instruction. To calculate h_{n+1}, equation (6-13) is multiplied by $\frac{1}{2}$:

$$h_{n+1} = \frac{1}{2} x_n + r_n * \frac{1}{2} x_n = h_n + r_n h_n \tag{6-16}$$

So, for one iteration step, in total three FMA instructions are needed, instead of four multiplications and one subtraction.

The initial value of $sqrt_n$ and h_n are calculated with the result of the reciprocal square root estimate $rsqrte$ as follows:

$$sqrt_0 = a * rsqrte \tag{6-17}$$

$$h_0 = 0.5 * rsqrte \tag{6-18}$$

To ease the precision proof, the input to the square root function could be reduced to an interval of 0.5 to 2.0. As the base two is used in computer systems, this can be easily done by an adjustment of the exponent. But this adjustment must be added to the result of the calculation on the reduced range afterwards, so from the execution time point of view, it is beneficial if the algorithm works on the whole range without a range reduction. So, in addition to the Gappa proof, it must be shown that there is no overflow in the calculations.

First, the initial values of $sqrt_n$ and h_n are checked regarding the range. Considering the uncertainties in the $rsqrte$ equation (6-17) resolves to:

$$sqrt_0 = a * rsqrte = \left(a * \frac{1}{\sqrt{a}} * \left(1 \pm e^{rel}\right) \right) = \sqrt{a} * \left(1 \pm e^{rel}\right) \tag{6-19}$$

The exponent range for the square root for a non-infinite positive double precision number is limited to $[-537{,}512]$. Consequently, as the absolute relative error of the estimate is way below 0.5, $sqrt_0$ will not overflow nor underflow, also considering the additional error introduced by the multiplication which is not considered in equation (6-19).

Below the result of the consideration of the uncertainty in $rsqrte$ for equation (6-18) is shown:

$$h_0 = 0.5 * rsqrte = 0.5 * \frac{1}{\sqrt{a}} * \left(1 \pm e^{rel}\right) \tag{6-20}$$

The exponent range for the reciprocal square root with the full positive non-infinite double number range as input is: [-512,537]. With the factor of 0.5, this range is reduced by one for both bounds. So, the range of the exponent for $0.5 * \frac{1}{\sqrt{a}}$ is [-513,536]. As above, with a relative

error of the estimate way below 0.5, h_0 neither overflows nor underflows. Especially as in this case, the multiplication can be executed without a rounding error.

Next the range of the introduced “residual” r_n, as introduced in equation (6-15), is checked. First the initial value r_0 is checked:

$$\begin{aligned} r_0 &= \frac{1}{2} - sqrt_0 * h_0 = \frac{1}{2} - \sqrt{a} * \left(1 \pm e^{rel}\right) * \frac{1}{2} * \frac{1}{\sqrt{a}} * \left(1 \pm e^{rel}\right) \\ &= \frac{1}{2} - \frac{1}{2} * \left(1 \pm e^{rel}\right)^2 \end{aligned} \tag{6-21}$$

As the initial value does not overflow, the values for the next iteration steps also do not overflow as the residual becomes smaller in each iteration step. For small relative errors this statement can only underflow, but an overflow cannot occur. The precision loss due to underflow is covered in the Gappa proof.

In order to show the precision of the function, Gappa is used. As Gappa cannot handle loops, the iteration of the Newton-Rapson method has to be unrolled for the input script to Gappa. Gappa is designed to generate proofs bounding the errors due to floating-point arithmetic. So Gappa does not know about the convergence of the Newton-Rapson method. This information can be passed by the following hint:

```
1  (fma(sqrtn0,0.5 - (sqrtn0 * hn0),sqrtn0) - Msqrt)/ Msqrt
   -> - (1.5 + 0.5 * err0) * (err0 * err0);
```

Listing 6-2: Gappa Code to add the Convergence Information

On the left-hand side of the expression in Listing 6-2, the calculation of the relative error after one iteration is written. This is done using the following components: The calculation, without rounding errors, performed during one Newton-Rapson iteration and the exact result of the square root Msqrt. On the right-hand side of the expression in Listing 6-2, the calculation of the new relative error based on the relative error of the previous / initial step is listed. These hints must also be “unrolled” and are therefore provided for each iteration step.

In order to get meaningful results for all possible inputs, the whole input range is split into several smaller intervals. Each interval contains all numbers with a certain exponent. The calculation time for these proofs is a few minutes.

It is possible to show that the absolute relative error of the square root is below $1b - 52$ after four iterations. Consequently, the result is within $\pm\, 1ulp$ to the actual result. In order to reach a precision of $\pm 1\; ulp$ for single precision numbers, only one iteration step could be omitted. Considering this, the creation of a separate function for the single variant is not a good tradeoff between the memory consumption of such a function and the saved execution time.

If a compliance to the IEEE 754 [3], which requires a correctly rounded square root, is mandatory, the result of the Newton-Rapson iteration is refined with the help of a final rounding check.

As proposed by [49], a Tuckerman test (see the page after next) is used to check whether the result of the Newton iteration is already correctly rounded to the nearest floating-point number or not. For this test the calculated square root is multiplied with its successor and predecessor floating-point number. Based on the relation to the original input and the results of the multiplications, the correctly rounded solution can be decided. The necessary successor and

predecessor can be calculated with the use of the ulp. To get the ulp, integer arithmetic is used to get the correct floating-point number. As first step, the ulp of the estimate is calculated. Here, only the estimates of the square root are handled. For that reason, subnormal numbers do not need to be considered here, as the square root of the smallest non zero double number has an exponent of - 537, which is still far away from the smallest exponent of normalized numbers - 1022. Thus, the ulp for all double precision numbers in these cases is defined as:

$$ulp(sqrt_n) = 2^{e-52} \tag{6-22}$$

Where e is the floating-point exponent of $sqrt_n$. No distinction of cases is needed. The code for calculating the corresponding ulp is listed in Listing 6-3:

```
1  p_exp = (CPU_INT16S*)(& sqrtn);
2  // -832 used instead of left shift 4 minus 52 left shift right
3  exp = (* p_exp);
4  exp = exp - 832;
5  exp = exp & 0xFFF0;
6
7  p_ulp = (CPU_INT16S*)(& ulp_gn);
8  ulp_gn = 1.0;
9  * p_ulp = exp;
```

Listing 6-3: Code for the ulp Calculation

The code works only for big endian systems like the target at hand. First, the 16 highest bits of the double number are extracted by setting an int16 pointer to the address of the square root estimate. Then, the content of this memory is assigned to an int16 variable. This variable then contains:

- The sign bit
- The exponent
- The four most significant bits of the significand.

Here, only the exponent is of interest. As the code is part of the square root calculation, the sign bit is always zero and does not matter. The significant bits are set to zero by the “bitwise and” in line five. In order to reduce the exponent by 52, 832 is subtracted. The reason why 832 is subtracted is that the exponents begins on the fourth LSB and 832 is equal to 52 shifted by four to the left. Otherwise the content of the variable could also be shifted four to the right, 52 can be subtracted and then the content can be shifted back four to the left. But this would lead to an additional instruction which increases the WCET. The result of the “bitwise and” is the exponent of $ulp(sqrt_n)$. This integer number then has to be converted back to a double precision floating-point number. The correct floating-point number has a positive sign, the calculated exponent and all bits of the significand need to be zero. This can be achieved by setting the corresponding bits with the help of integer pointer and interpreting the memory as double precision floating-point number afterwards. Here, an alternative solution is implemented. A floating-point number is set which fulfills both a positive sign and an empty significand. 1.0 is chosen as this is an easy number. After setting the value of the floating-point number, the exponent of the number is manipulated by getting the address of the floating-point number to an int16 pointer. The dereferenced pointer is then set to the calculated exponent. So, after execution of line nine in Listing 6-3, the variable ulp_gn contains the correct ulp. With this ulp then the successor and predecessor floating-point number can be calculated. The calculation of the successor is a simple addition of the ulp and the $sqrt_n$. In case of the predecessor, the special case when all bits of the significand of $sqrt_n$ are zero has to be considered. This correlates to a significand of 1.0. In this case the value of the predecessor’s

exponent is one less. Therefore, also the ulp of the predecessor is half the size. So the predecessor is calculated as follows:

$$x^{-} = \begin{cases} x - ulp(x)\; for\; the\; significand\; of\; x \neq 1.0 \\ x - 0.5 * ulp(x) forthe\; significand\; of\; x = 1.0 \end{cases} \tag{6-23}$$

A check if all significant bits of the inputs are set to zero would imply integer arithmetic and interpreting the memory different. This can be omitted by first subtracting only half of the calculated ulp, which can be performed with a fused negative multiply subtract instruction. In case all the bits are zero, the actual predecessor is calculated in other cases due to rounding the result equals the output. Therefore, if the result is equal to the input, one whole ulp is subtracted.

The successor and predecessor can then be used to execute the Tuckerman test, which is defined as: *"If a and r are floating-point numbers, then r is $\sqrt{a}$ rounded to the nearest if and only if*

$$r\left(r - ulp(r^{-})\right) < a \leq r\left(r + ulp(r)\right) \tag{6-24}$$

where r^{-} is the floating-point predecessor of r" [49].

For the actual calculation of this test, a is brought to the other side of the inequality and then the result is checked for zero. Thus, the calculation can be performed using fused subtract multiply instructions. If one of the inequalities is not fulfilled, the correct rounded result is either the successor or the predecessor. Dependent on which inequality fails, the correct result can be set and returned afterwards.

The calculation of this Tuckerman test is not negligible regarding the timing of the function. The WCET without a test and correction of the result is 182 cycles. With the test and the correction of the return value, the WCET is increased to 291 cycles. This is a plus of approximately 60 %. Considering the fact that the square root is called 223 times on the WCET path of the complete software, this is quite a high effort to get the last bit accuracy. Especially as all other elementary functions normally are not correctly rounded, which is in particular true for the implementations in this thesis. As a result, for the integration to the complete software the final rounding test is not integrated in the square root function. So, the compliance to IEEE 754 [3] is not completely given in this point.

In the application software, the square root is often used to calculate the length of vectors to calculate the corresponding unit vector. So in those cases, the user is actually not interested in the square root but in the reciprocal square root. A review of the application software example at hand showed that in approximately 43 % of the cases where the square root is calculated, the reciprocal is needed. Although, Simulink provides the possibility of a reciprocal square root setting for the square root block [90], there is no standard ANSI C [47] library function for the reciprocal square root. This is probably also the reason why the SLCI does not support this setting [91]. Next to the application software, also the arcus sine based on the arcus tangent needs a reciprocal square root. (Compare section 5.4.2)

As shown above, during calculation of the square root also $\frac{1}{2} * \frac{1}{\sqrt{a}}$ is calculated. By multiplying with two, the reciprocal of the square root can be calculated. As shown in section 4.2, the latency introduced by a multiplication is much lower than the one by a division. Therefore, the multiplication would be the preferred way to calculate the reciprocal of the square root. Such a

function could be provided to the model-based application by a special legacy code function block. This way the compatibility to the SLCI could be kept.

6.2 Function Stub for the Runtime Error Analysis

As the square root is defined in the IEEE 754, Astrée provides an implementation of the square root. In case the square root is integrated with the rounding test and correction, the function provided by Astrée can be directly used. In addition, the fact that the implementation at hand limits negative values to zero still needs to be considered. This can be achieved by an “if” statement in the stub. In order to ensure that this statement does not hide true runtime errors, the “if” statement checks that the input is negative and still close to zero. If this condition is true the return value is set to zero. For negative input values not close to zero, an alarm is raised. For all positive input values, the return argument is calculated by the Astrée square root. For the function without the rounding test and correction an increased error has to be considered in the runtime error analysis. This can be achieved by multiplying the result of the Astrée square root with a variable in the range $[1 - 2^{-52}, 1 + 2^{-52}]$.

7 Exponentiation

In general, the ANSI C defines the power function with both arguments, the base and the exponent, as floating-point numbers [47]. To implement such a function, it can be transferred using the following equation [11]:

$$x^y = 2^{y*log2(x)} \tag{7-1}$$

The equation relies on the implementation of a logarithm to the base of two and an exponential function. For both of these sub-functions, a range reduction can be implemented based on the fact that the base of the number system is two. But on the reduced range the approximation of these functions is still quite complex. For the software at hand, the logarithm function is not further used, so an implementation based on equation (7-1) might lead to a huge effort.

7.1 Function Implementation

As the flight control software uses the power function only with integer exponents, an implementation of the power function also supporting floating-point exponents might not be necessary. Therefore, here a special version of the power function is introduced only supporting integer exponents. The range of the exponent is restricted to $[-128,127]$, which correlates to a signed integer with 8 bit. This range is sufficient for the software at hand. For an integer exponent n, the calculation can be easily broken down into n multiplications for $n > 0$ or into n divisions for $n < 0$. In case the absolute value of n is huge, this is not very efficient. Especially as the division is an instruction with a huge latency, a negative n would lead to highly inefficient code. In addition, each instruction might introduce a rounding error, and so, the overall rounding error of n multiplications / divisions might not be negligible.

In order to omit such problems, the chosen implementation explicitly executes multiplication and division but by the use of associative properties the number of instructions are reduced. Utilizing this approach, the execution time and the rounding error can be decreased. As this implementation is an exact mathematical representation, no approximation error occurs.

As divisions have a higher computational effort than multiplications, division shall be omitted. To reduce the numbers of divisions in the case of a negative exponent, the following equation is used:

$$x^n = \left(\frac{1}{x}\right)^{-n} \tag{7-2}$$

Therefore, only one division is necessary at the beginning and all the consecutive instructions can be performed as multiplications. So without loss of generality, in the following steps only positive values of the exponent n are considered.

In order to reduce the number of multiplications, the exponent is split into a sum of integer powers of two. If n_i correlates to the value of the bit i of the integer representation n and $i = 0$ represents the LSB the exponent can be written as:

$$x^n = x^{n_7*2^7+n_6*2^6+\cdots+n_0*2^0} = x^{n_7*2^7} * \dots * x^{n_0*2^0} \tag{7-3}$$

The term $x^{n_i * 2^i}$ resolves to 1.0 in case the bit i is not set and to x^{2^i} in case the bit i is set. The multiplication with 1.0 has no effect and can be neglected. The implementation of equation (7-3) into C code is straightforward. To get all set bits of the exponent, a loop is implemented which is executed as long as the exponent is greater than zero. At the end of the loop, the exponent is shifted right by one. Consequently, in the ith iteration of the loop, the LSB of the modified exponent correlates to ith bit of the original exponent. In case the LSB is set, which is determined by a “bitwise and” with one, the return value is multiplied by x^{2^i}. In the case the LSB is not set, the return value is not modified. In order that applicable x^{2^i} is available, the current x^{2^i} is squared by a simple multiplication in each iteration of the loop to get $x^{2^{i+1}}$ for the next iteration. In order to get the correct result, the return value is set to 1.0 prior to the loop. This ensures also a correct return value in case the exponent is zero. The complete code of the loop is shown in Listing 7-1:

```
1  while(exp_abs > 0)
2  {
3    if((exp_abs & 0x01) != 0)
4    {
5      ret_val = ret_val * xn;
6    }
7    xn = xn * xn;
8    exp_abs = exp_abs >> 1;
9  }
```

Listing 7-1: Loop for the Calculation of the Power Function

In order to proof the precision of the algorithm, Gappa scripts are compiled. As Gappa does not support loops, first the loop has to be unrolled. Only in case the “if” statement evaluates to true an error is introduced. So in case that an upper bound of the error shall be determined, it is sufficient to consider the “if” statement always true. In order to get the worst case, it would be sufficient to consider only the case of the highest exponent. But due to the chosen implementation, the number of instructions and the consecutive rounding errors grow with the exponent. For that reason, considering only the maximum exponent leads to a huge overestimation in case of small exponents. In order to give a better bound for the error, the following exponents are considered separately: 3, 7, 15, 31, 63, and 127. As these represent numbers where all consecutive low bits are set, they give the worst case and the precision of the intermediate values can be estimated. As the proof can only be successfully compiled in case the result does not over- or underflow, the input range of the different proofs is also adapted accordingly. Therefore, the separate proofs also bring a benefit in order to show the validity over an extended input range for smaller exponents. Especially as the function return value grows exponentially. The structure of the Gappa script is as presented in section 4.1. The results for the different positive exponents are given in Table 7-1.

Exponent	Input Range	Relative Error e^{rel}
3	$[2.8 * 10^{-103}, 4.5 * 10^{102}]$	$2^{-52} = 2.2 * 10^{-16}$
7	$[1.1 * 10^{-44}, 9.8 * 10^{43}]$	$2^{-49.3} = 1.4 * 10^{-15}$
15	$[3.1 * 10^{-21}, 3.4 * 10^{20}]$	$2^{-48.6} = 2.3 * 10^{-15}$
31	$[1.2 * 10^{-10}, 8.6 * 10^{9}]$	$2^{-48} = 3.4 * 10^{-15}$
63	$[1.3 * 10^{-5}, 7.7 * 10^{4}]$	$2^{-47} = 6.8 * 10^{-15}$
127	$[3.8 * 10^{-3}, 266]$	$2^{-46} = 1.4 * 10^{-14}$

Table 7-1: Precision of the Power Function

The input ranges in the table above are chosen such that the result is still in the range of normalized numbers. Only in those cases the relative error correlates to a number of potential

wrong consecutive LSBs of the significand. For input ranges which lead to de-normalized numbers, extra proofs are compiled. Instead of the relative error, the absolute error is considered, as de-normalized numbers have a constant resolution. The maximum error here is in the range of $2^{-1067} = 1.8 + 10^{-322}$.

The results above are for positive exponents. For the negative exponents, also the inversion prior to the loop must be considered. Due to this additional instruction, in this case the result precision can be approximately one bit lower than for the positive exponents. But still the reached precision is high.

In the trajectory generation, the power function is often used to calculate a polynomial. Instead of using the power function to calculate each x^n, the Horner scheme can be applied. In this case the polynomial would be evaluated by multiplications with x and additions. The sequence of evaluation is handled by brackets such that the result is mathematically identical to the classical form with exponents of the input value. As shown in Figure 7-1, it is also possible to implement the Horner scheme for a specific degree in a Simulink model/library.

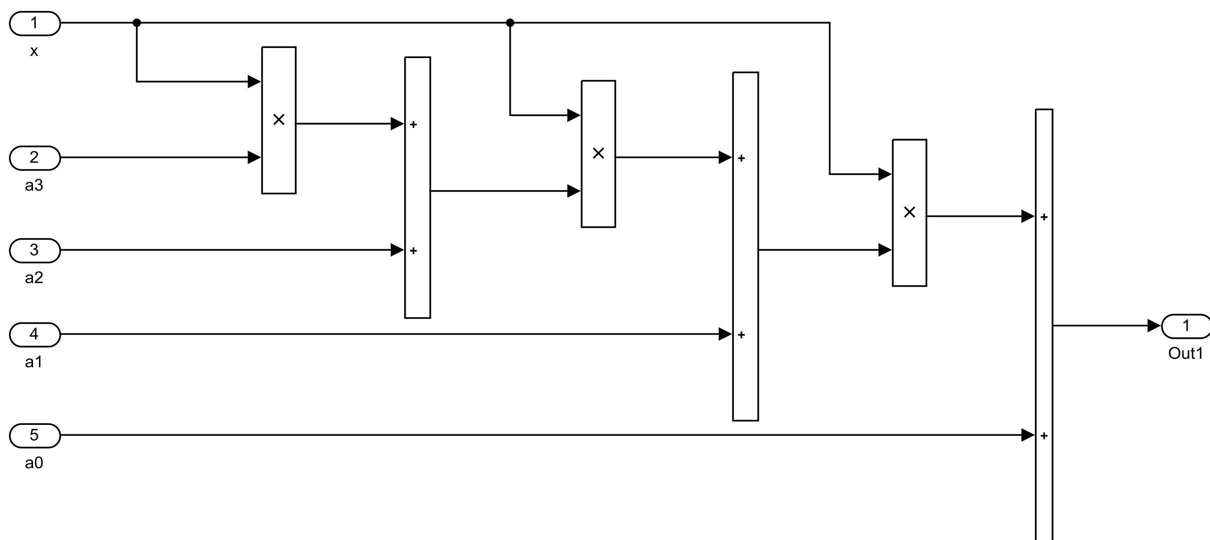

Figure 7-1: Simulink Implementation of a Polynomial in Horner Scheme

The benefit of such a solution is that fewer instructions are necessary and so, the execution time will be lower. The model with the consecutive multiplications and additions gets quite confusing for higher polynomial grades. As an alternative solution, Simulink provides a polynomial block [90]. But as the coefficients must be known a priori, this block is not useful for the current use case. Alternatively, the DSP System toolbox provides a Polynomial Evaluation block, where the coefficients can be adapted during the runtime [92]. But this block is not supported by the SLCI [91]. To omit the code inspector problem, a legacy code function can be implemented. To make the function as generic as possible, a variable degree of the polynomial could be handled by a pointer to an array containing the coefficients as function parameter. Additionally, then the length of the array, and the input value to the polynomial must be passed to the function as parameters. The function body can implement a for-loop, which multiplies and adds all the components sequentially. Another benefit of the implementation in C source code is that a FMA instruction can be used in case it is present on the applicable CPU. This will reduce the execution time approximately by a factor of two in comparison to the implementation by a Simulink model.

7.2 Function Stub for the Runtime Error Analysis

Due to the fact that a small input to the exponential function can lead to a very huge output, stubbing this function is quite difficult. In order to omit this problem, the actual use case of the power function is taken into to considerations. For the flight control software at hand, the power function is mainly used in the ATOL and the trajectory generation module. In the trajectory generation module, the power function is often applied during the calculation of the transition maneuver [60, 61]. In the ATOL module the trajectory generation is based on Bézier curves, and in this context the power function is used there [93].

In the trajectory generation system, the absolute value of the base is in many cases smaller or equal to 1.0, and the exponent is always bigger than one. In these cases, an easy stub can be developed. In case the base is positive, the range of the output value can be set to $[0.0,1.0]$. This is valid as numbers smaller or equal to 1.0 multiplied by its own, never will get bigger than 1.0. In case the base argument is smaller than zero, in addition it must be considered that the sign will change for even exponents and be the same for odd exponents. To cover this, in this case the return range is simply set to $[-1.0,1.0]$.

As the implemented power function is limited to integer exponents, the function could be actually stubbed by unrolling the multiplications / divisions in a “for loop”. Although this is not an option for the target implementation for the stub such an approach would be acceptable. As in the analysis with Astrée all possible rounding modes are considered for the calculation [77], the result will stay sound. But still the problem exists that the result can get quite large very soon.

With this approach it was possible to show that the return values do not overflow. But for the complete calculation of the Bézier curves in the ATOL system, the result was such unprecise that in the following calculation false alarms were raised. An approach to omit this problem can be found in chapter 8.

8 Runtime Error Analysis

As described in section 4.1, a runtime error analysis is performed in order to show that the design ranges of the elementary functions are not violated. For the analysis the tool Astrée is used. In contrast to other tools of this category, Astrée provides a powerful set of annotations which can be used to increase the precision of the analysis.

The runtime error analysis shall be applied to the integrated C source code, containing the complete auto generated code and the developed functions. In order to get valid and sound results, it must be guaranteed that the operation mode is similar to the real application. Therefore, the analysis also must consider the initialization phase followed by the cyclic execution of the step function. Especially if the repetitive behavior is neglected, this might lead to undetected errors, like for example an overflow of internal counters. Next to the cyclic execution, it must be ensured that the whole possible input range is assigned to the inputs of the analyzed code. These points are ensured by a wrapper function generated in Astrée. The wrapper contains a specific main function. There, first a call to the initialize function is executed, afterwards, the step function is called cyclically. In advance of the call of the step function, the applicable inputs are written. This is done by a duplicate of each input datum, which is defined as __ASTREE_volatile_input [77], with the range according to the interface control document. Once per cycle this duplicate is then assigned to the actual input signal. Currently, the wrapper is generated manually. But as the wrapper is stored in a simple text file with the syntax similar to C, it also could be generated automatically, for example with an extension of the interface generator used for the embedded code.

The version of the code without any annotations or rearrangement of the model produces more than two thousand alarms. In order to reduce the number of alarms, two different approaches are applied. Some parts of the model are re-implemented and in other cases, the annotations provided by Astrée are used. The reimplementation is mainly used for model libraries. The reimplementation does not affect the functionality, it only has the purpose to lead to a more clear control flow at source code level. For example, exclusive branches in the model are modeled such, that the exclusive branches are actually implemented by exclusive statements in the C source code. More details on the adaptation of the model can be found in the following subchapter. The annotations are used in cases where the problems do not depend on the way the model is built, but more on the way the code is generated and also in the cases where only one part of the code is affected or the correlations are too complex for the abstract interpretation. Most of the annotations are partition control annotations. These annotations increase the precision of the analysis and cannot lead to a result, which is not sound. In the ATOL and trajectory generation module also some ranges need to be annotated. If a too tight range is given in such an annotation, potential unsound results may be produced. To omit such problems, the ranges are either motivated by further mathematical analysis or by an argumentation from system perspective. More details can be found in section 8.2. Applying these methods lead to a two-digit number of alarms for the analysis of the whole application software. Most of the remaining alarms are false alarms, as some constructs cannot be analyzed. The remaining alarms might only occur in cases of wrong sensor data or during non-nominal operation of the aircraft. Thus, they will not have an impact during normal operation. Therefore, the alarms are considered as acceptable.

8.1 Adaptation of Model

In order to analyze the code, Astrée uses the method of abstract interpretation [77]. By applying this method, some information is abstracted in order to reduce the problem such that it can be solved with normal computational power. The abstract interpretation uses several domains in order to get a sound result that is still precise with a low number of false alarms.

Also with the use of several domains, it is difficult to detect branches, which are implemented independently but actually depend on each other. This is often the case when switches are used at model level. In case the dependency is not detected, this normally leads to a too wide range of the value of variables, which can lead to a number of false alarms. This problem can be either resolved by annotations of the specific sections or by a different implementation. As the source code is automatically generated out of the model, for a change of the implementation the model must be adapted. In case the problematic construct is contained in a model library or reused subsystem, the adaption of the implementation can be easily performed on the model level. Either the corresponding library can be adapted or the system can be replaced in all applicable caller models by a script with an improved version of the original model. Therefore, with this method adaptions can be provided to the complete code without a high manual effort. For that reason, this method is preferred to the manual annotations of all occurrences. This is especially of interest as some functionality, gathered in reused models at model level, is actually in-lined at source code level. Which means on source code level several locations must be adapted but on model level it is sufficient to solve it once.

An example for a code which is difficult to analyze is the block “Saturation Dynamic” of the default Simulink library. This block is actually a masked subsystem, which consists out of more basic blocks. The implementation is shown in Figure 8-1.

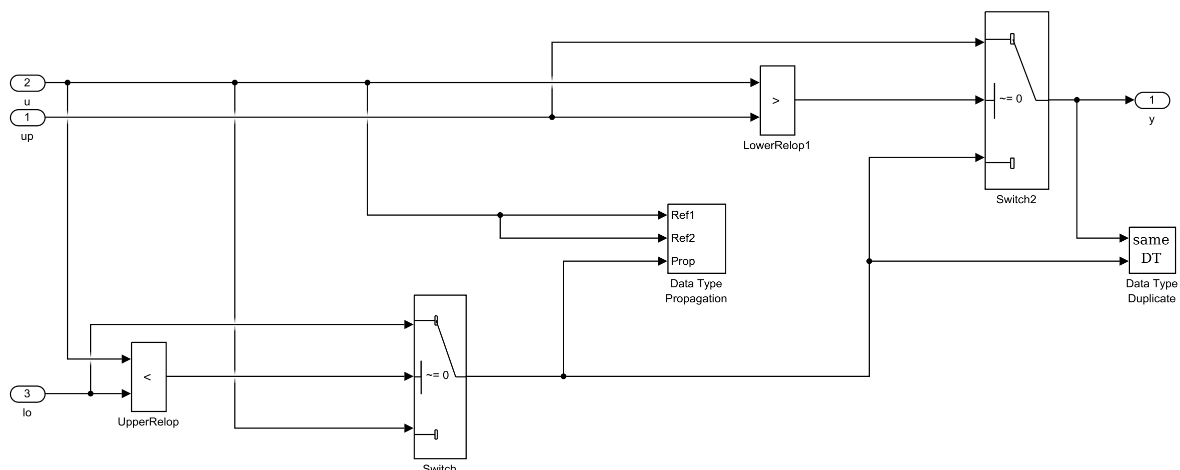

Figure 8-1: Implementation of the Default “Saturation Dynamic”

The two switches lead to the following construct in the source code:

```
1   if(b_u < b_lo)
2   {
3     b_Switch = b_lo;
4   }
5   else
6   {
7     b_Switch = b_u;
8   }
9
```

```
10  if(b_u > b_up)
11  {
12    y = b_up;
13  }
14  else
15  {
16    y = b_Switch;
17  }
```

Listing 8-1: Code Generated for the Block “Saturation Dynamic”

The problem of the construct above is that the input to the second “if else” is again the signal `b_u`. During the analysis, the correlation between `b_u` and `b_Switch` is abstracted. This means during the abstract interpretation it is not obvious that in the case that `b_u` is bigger than `b_up` also `b_Switch` is bigger than `b_up`. Consequently, the possible values of `b_Switch` in line 16 are not further restricted during the analysis. This leads to the fact that the limitation to the upper bound has no effect in the analysis, with the consequence that the analyzed range of `y` might be too big. In order to omit this problem, the block is re-implemented using action sub-systems. The new implementation is shown in Figure 8-2

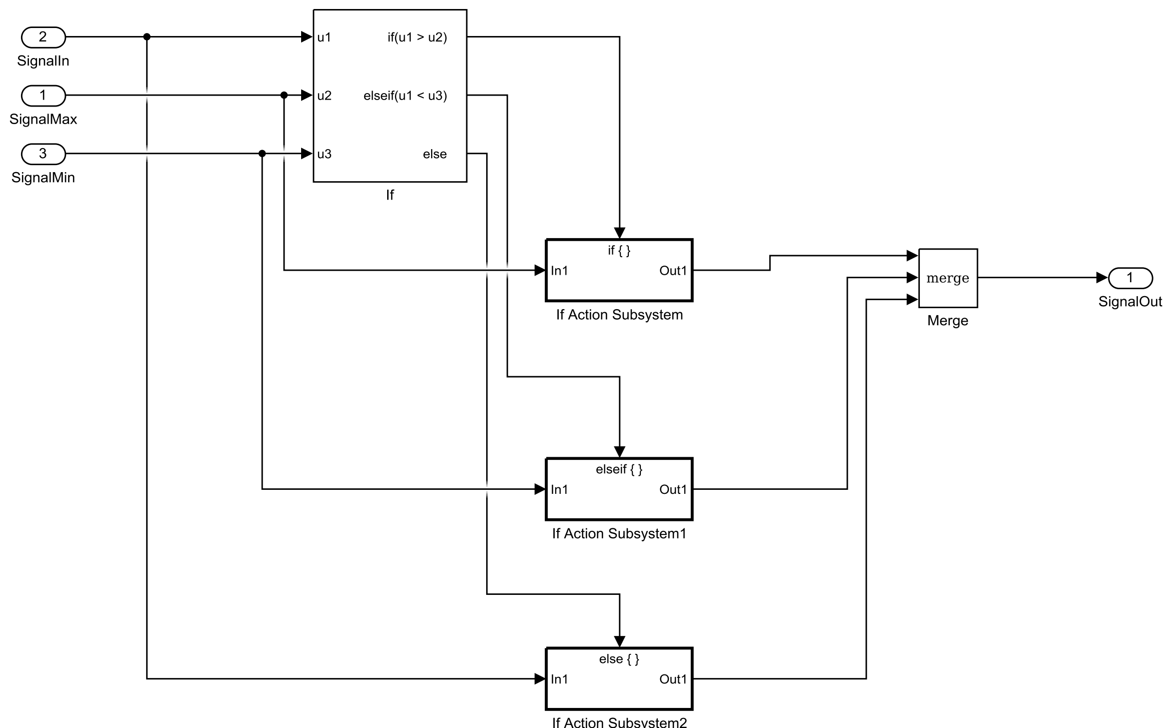

Figure 8-2: Implementation of the Custom “Saturation Dynamic”

The code generated for the modified dynamic saturation is shown in Listing 8-2:

```
1   if(SignalIn > SignalMax)
2   {
3     SignalOut = SignalMax;
4   }
5   else if(SignalIn < SignalMin)
6   {
7     SignalOut = SignalMin;
8   }
9   else
10  {
11    SignalOut = SignalIn;
12  }
```

Listing 8-2: Code Generated for the Custom “Saturation Dynamic”

For the given example it would also be possible to solve the problem by just changing the input to the block ”LowerRelop1” to the already switched signal. But the benefit of the implemented version is that the code resolves to three exclusive implemented branches, which makes the understanding easy and also leads to benefits in the execution time analysis. In addition, this is also beneficial if the signal of the comparison shall be further used. A possible implementation of such a case could be to first calculate Boolean signals based on the comparison of the limits with the input. A second step limits the signal based on the Boolean values. Therefore, in order to produce a precise result, the runtime error analysis must detect the dependency between the Boolean and the data range of the input. If this dependency is abstracted, the imprecise result might lead to false alarm. In the case of an implementation with action subsystems, the corresponding indication can be set within the separate subsystem in conjunction with the actual limitation of the input. As all the branches are exclusive, in this case it is sufficient if the interval domain is used for the abstract interpretation. One little downside of the implementation with action subsystems is that the functionality potentially is not directly obvious to the user as some functionalities are hidden in the subsystems. As the changed systems are mainly library functions and common function blocks, whose functionality shall be clear a priori without looking at the actual implementation, this is acceptable.

Additionally to the change of the dynamic saturation library, similar changes are applied to the following function of the common compliant lib developed at the institute:

- prot_div: Protected division divide by alternative value or divisors with small absolute value
- sat_w_flg: Dynamic saturation with an indication flag
- sat_w_diff: Dynamic saturation with calculation of difference from limit
- sat_w_2flg: Dynamic saturation with separate flags for upper and lower limit

In all these functions, the relevant switches were replaced by action subsystems. In case of the protected division also the internal threshold, which should protect wrong usage of the block, was removed. This internal threshold limits the alternative divisor. As the runtime error analysis would detect a limit that is too small, such an internal limitation is not necessary.

Beside these changes on commonly used functions, a few changes were applied to the trajectory generation module. The changes are either the simplification of a calculation or changes of the control flow as the branches sometimes rely on Boolean signals, which correlate to certain range of the local inputs. As the calculation of the Boolean might be separated from the actual calculation, the information that the branches are only executed for certain ranges is abstracted.

An example for a calculation change is the calculation of the radius in the case of a fly over [61]. The original implementation is as follows:

$$b = \arccos\left(\frac{1 - sin(a)}{0.5}\right) \tag{8-1}$$

The signal b is then limited to a maximum value of $b_max = \pi/3$:

$$b' = \begin{cases} b & if\ b < b_max \\ b_\max & othewise \end{cases} \tag{8-2}$$

In the following steps, the cosine of the limited signal is used:

$$c = \frac{\sin(a) + \cos(b')}{1 - \cos(b')} \tag{8-3}$$

In the implementation for the division, the protected division block is used. But as the threshold to zero of the division is $1e-10$, in case of the abstract interpretation c will take values up to $2e10$. As big ranges propagate through the following calculations, the high range might lead to overflows. To omit this, the calculation is reformulated to get a mathematically identical result but with a tighter bound in the analysis.

The calculation basically has two different branches: One where b is limited to b_max and one where b is not within the limit. As the calculation in equation (8-1) only depends on a or rather on $\sin(a)$ it is easy to switch the calculation dependent on value of $\sin(a)$. To simplify the calculation, the two branches are considered independently: First the case where b is limited is considered. In this branch $\cos(b')$ resolves to the constant 0.5. If this is applied to equation (8-3) the result is:

$$c = \frac{\sin(a) + 0.5}{1 - 0.5} = 2 * \sin(a) + 1.0 \tag{8-4}$$

In the case that b is not limited, the calculation of arcus cosine and immediately calculating the cosine of the signal cancel each other for the applicable range. Therefore, the calculation resolves to:

$$c = \frac{\sin(a) + \frac{1 - sin(a)}{0.5}}{1 - \frac{1 - sin(a)}{0.5}} = \frac{2 * \sin(a) + 1 - \sin(a)}{2 - 1 + \sin(a)} = 1 \tag{8-5}$$

With this implementation, the range in the analysis can be easily reduced to an upper bound of three, which is significantly smaller than the one with the original implementation. The author of the algorithm originally chose the first implementation, as it is more closely related to the derivation of the algorithm. But as the Simulink model is also used for generation of the embedded code, a tradeoff must be found between a model, which is easily understandable and an implementation, which fits the needs of a safe embedded system. Such needs are for example the compliance to performed analysis steps, but also the numerical performance and an efficient implementation.

An example where the analysis was improved by the change of the control flow is the calculation of the parameters that define a clothoid augmented “radius to fix maneuver”. Input to the system is a special case flag. The calculation of the signal r_circle_hor_m differs, dependent on the value of the flag. In case the flag is false, the calculation contains a division by $\sin(\alpha/2)$, which is not contained in the case that the flag is set to true. As the division is protected, no alarm is raised directly here. But as described in the previous example, the signal can get quite high values. To improve the analysis at this point, a manual analysis of the complete integrated software is executed to find a correlation between the special case flag and α. All call sites of the function are analyzed regarding the condition when this flag is set. This shows that the flag is always set to true if the signal α gets smaller than one degree. Thus, the flag and the range of the signal α are directly linked. As write and read of the flag are quite far spread over the software, no octave domain is generated in the analysis of the software and so the range of α is not further restricted in the special case true branch. To solve this issue, either the range of α can be manually annotated in the different branches. Or the

implementation can be changed in a way that the branching is directly depended on the range of α. Both variants lead in conjunction with the improved sine stub to a much tighter range in the analysis. In order to reduce the manual effort on the analysis side, here the way to change the model was chosen.

8.2 Improved by Annotations

Additionally to the changes of the model, some constructs were annotated in order to omit false alarms.

One of the constructs which need to be annotated, is the multiport switch in case the same action is performed in more than one state. For each multiport port switch, a “switch case” statement is generated in the source code by the Embedded Coder. In most occurrences, each case contains a simple assignment. If in two or more cases the same action shall be performed, instead of directly performing this action in the case of the “switch case”, a Boolean “guard” is generated. This guard is set to false in advance to the “switch” statement and set to true in the corresponding cases. After the “switch case” construct an “if” statement that is dependent on the guard follows. In the “if” branch then the actual statement is performed. In case a variable is initialized during this switch case statement and a guard is used, a false alarm “Use of uninitialized variables” will occur. The reason for that is that the dependence of non-exclusive implemented branches is normally abstracted by the analysis. Therefore, it is not clear for the analyzer that the “if” is true in the case the variable is not written during the execution of the “switch case” statement. This can be omitted by a partition control directive [77] before the switch and the “if” statement. If this directive is added, the exclusivity of the different branches is correctly considered in the analysis. After the “if” a partition merge directive is introduced to merge the branches again. As each partition control increases the computational effort of the analysis, the merge ensures that the different branches are joined as soon as possible. For that reason, the analysis can still be solved with a reasonable computational effort. Also, cases with more than one “guard” are contained in the software at hand. The approach here is the same: Before the “switch” statement and for each “if” statement a partition control directive is introduced and after the last “if” condition the merge directive follows.

One specialty of the analysis is that none of the shared libraries generated by the Embedded Coder are stubbed. For example, in the default configuration of a Polyspace Code Proofer project for generated code the lookup functions are stubbed [94]. The reason probably is that the binary search, contained for example in the table lookup functions, is quite difficult to be analyzed with the abstract interpretation. Therefore, in the analysis with Astrée, some annotations were necessary in order to get no false alarms in the algorithm itself and to get the correct range for the return value. As example, here the annotations made to the function look1_iflf_binlca are shown. The code of other lookup tables is annotated in a similar way. The function look1_iflf_binlca is the implementation for the lookup table with the following settings:

- Lookup dimension: one
- Search method: binary
- Interpolation method: linear
- Extrapolation method: clip
- Use previous index: off

The source code is shown in Listing 8-3:

```
1  real32_T look1_iflf_binlca(real32_T u0, const real32_T bp0[], const real32_T
2    table[], uint32_T maxIndex)
3  {
4    real32_T y;
5    real32_T frac;
6    uint32_T iRght;
7    uint32_T iLeft;
8    uint32_T bpIdx;
9
10   if (u0 <= bp0[0U]) {
11     iLeft = 0U;
12     frac = 0.0F;
13   } else if (u0 < bp0[maxIndex]) {
14     bpIdx = (maxIndex >> 1U);
15     iLeft = 0U;
16     iRght = maxIndex;
17     while ((iRght - iLeft) > 1U) {
18       if (u0 < bp0[bpIdx]) {
19         iRght = bpIdx;
20       } else {
21         iLeft = bpIdx;
22       }
23
24       bpIdx = ((iRght + iLeft) >> 1U);
25     }
26   __ASTREE_partition_begin((iLeft));
27   __ASTREE_known_fact((u0 >= bp0[iLeft] && u0 <= bp0[iLeft + 1U] ));
28     frac = (u0 - bp0[iLeft]) / (bp0[iLeft + 1U] - bp0[iLeft]);
29   __ASTREE_partition_merge_last(());
30   } else {
31     iLeft = maxIndex;
32     frac = 0.0F;
33   }
34   if (iLeft == maxIndex) {
35     y = table[iLeft];
36   } else {
37     __ASTREE_partition_begin((iLeft));
38     y = table[iLeft] + (frac * (table[iLeft + 1U] - table[iLeft]));
39     __ASTREE_partition_merge_last(());
40   }
41
42   return y;
43 }
```

Listing 8-3: Code of the Simulink Utility Function “look1_iflf_binlca”

First for the calculation of the fraction and for the calculation of the actual return value, a “partition begin” annotation according to `iLeft` is introduced. Otherwise, for the calculation of the fraction a false alarm, indicating a division by zero, will occur. The reason is that for `bp0[iLeft + 1U]` and `bp0[iLeft]` the whole possible range will be assumed and so potentially the same index is accessed, so the difference is potentially zero. With the partition according to iLeft, for the analysis it will be clear that two contiguous elements of the breakpoint array are accessed. Consequently, in case of an even space lookup table the divisor will be correctly analyzed as constant value. In case of not even spaced breakpoints, the corresponding range will be correctly analyzed. For the calculation of the return value, the opposite case leads to an increased loss of precision. So without the partition annotation, `iLeft` will be in the range from zero to `maxIndex - 1` and `iLeft + 1U` will be in the range from one to `maxIndex`. So for example for monolithic increasing lookup values, the difference from the last to the first lookup table would be possible in the analysis, which could be much too big. Similar to the annotation by the calculation of the fraction, here the annotation ensures that two contiguous lookup values are accessed and thus the correct range of the difference

is analyzed. In case that the range of the fraction is correct in the analysis, then the analyzed range of the return value is in the correct range. To achieve a correct range of $[0.0,1.0]$ for the fraction, in the analysis a third annotation is necessary. In the analysis it is not recognized that after the binary search is finished `iLeft+1` equals `iRght`. Therefore, it is also not clear that the following condition is true: `u0 <= bp0[iLeft + 1U]`. Also, the fact that `u0 >= bp0[iLeft]` is fulfilled is not recognized in the analysis. This might especially be difficult as this condition can only be derived implicitly from the "else" branch in the binary search and the fact that the breakpoints of a loockup table must be monolithically increasing. In order to improve the analysis, the third annotation provides the two conditions to the analysis by using a known fact annotation [77]. In conjunction with the partition annotation, in the analysis the range of `frac` is correctly analyzed to $[0.0,1.0]$. For that reason, it is possible to include the lookup tables in the analysis without the use of specific stubs. But one false alarm regarding the binary search still occurs in the analysis. For the condition of the while loop `iRght - iLeft`, an overflow in arithmetic alarm occurs. The reason is that the analysis is not aware of the fact that the `iRght` is always bigger than `iLeft`, which a manual review of the code shows. Unfortunately, also with different annotations it was not possible that the code is correctly analyzed. As a result, a false alarm occurs in the lookup algorithm, but as it does not impact the ranges on the return value, this is acceptable. The benefit of not stubbing the lookup tables is that the code is also part of the analysis and runtime errors can be detected. The downside is that due to the partition annotations, the computational effort will increase, which will lead to a longer computation time of the analysis.

Some annotations are also necessary, not because of the way source code is generated by the Embedded Coder, but due to the chosen implementation at model level.

The implementation for the input monitoring first executed several checks, which result in Boolean flags. Each flag represents the result of a certain check. All flags of the different checks are joined to an overall result by logical operators. Based on this, the function mn_LatSig forwards the input signal to the controller or latches the signal to an initial value or the last valid signal. The fact that the flags are calculated independently from the actual limiting of the signal, results in the circumstance that, during the abstract interpretation, it is not recognized that the function mn_LatSig actually performs a limitation of the signal. For the inputs which are transmitted via a scaled fixed-point value, this is actually not a problem as the limits of the range check are actually exactly the same as the possible range of the descaled value or close to it. The analysis wrapper "__astree_main_" considers the possible range of the descaled inputs, so the limitation has no or only a small effect in this case. Here, the range check just detects wrong behavior of the hardware. For signals which are actually transmitted as floating-point signals, this is different. As those signals in principle can have every possible value of the floating-point value including INF and NaN, here the limiting is essential for the following calculations. Therefore, after the function mn_LatSig with the corresponding signal as parameter, a known range directive is introduced, which limits the signal to the limits of the range-check. This range always considers the initial value. The annotations are mainly necessary for the signals of the air data computer and of the different actuator control electronics. A list with all annotated signals is given in Table 8-1:

Signal	Annotation
Dynamic pressure	__ASTREE_known_range((b_p_dynamic_ADS_meas_NDm2,[0;6000]));
Static pressure	__ASTREE_known_range((b_p_static_ADS_meas_NDm2,[147;105000]));
H Pressure	__ASTREE_known_range((b_h_pressure_ADS_meas_m,[-300;15000]));
Calibrated airspeed	__ASTREE_known_range((b_CAS_ADS_meas_mDs,[0;100]));
True Airspeed	__ASTREE_known_range((b_TAS_ADS_meas_mDs,[0;120]));
Alpha	__ASTREE_known_range((b_alpha_A_R_B_ADS_meas_deg,[-25;25]));
Beta	__ASTREE_known_range((b_beta_A_R_B_ADS_meas_deg,[-25;25]));
Aileron Actuator Turn Angle	__ASTREE_known_range((b_aildt_ace_1_deg,[-30;35]));
Elevator Actuator Turn Angle	__ASTREE_known_range((b_eldt_ace_1_deg,[-22;25]));
Rudder Actuator Turn Angle	__ASTREE_known_range((b_ruddt_ace_1_deg,[-30;38]));
Rudder Trim Turn Angle	__ASTREE_known_range((b_rtrim_ace_1_deg,[-30;30]));
Thrust lever Actuator turn angle	__ASTREE_known_range((b_athr_ace_lh_deg,[-6700;-150]));
Flaps Position Sensor	__ASTREE_known_range((b_flaps_position_meas_deg,[-5;50]));

Table 8-1: Signals Annotated after the mn_LatSig Function

Some parts of the software are annotated by introducing additional stubs. Especially the second order filter leads to problems in the analysis. This filter is used in the input monitoring and in several subsystems as command filter. In difference to other tools, Astrée provides a special domain to proof second order filters. But as this domain is based on abstract parameter matching [77], it does not work with the implementation at hand, especially as the filter also provides the possibility to be reset based on an external signal. In order to omit such problems, the original function call is replaced in the analysis using the substitute function feature. In the system design process, the filter coefficients are chosen such that the behavior is aperiodic without overshoot. As a result, in the stub only an assignment from the input to the output of the filter is contained. So, the range of the input and the output signal matches, this is in-line with the actual behavior of the filter.

A combination of an annotation and a change of the model is used for the Simulink subsystem, which is used to calculate the unit vector of an input vector. In order to calculate the unit vector, the length of the input vector is calculated using a square root. All components of the vector are then divided by the length of the vector, resulting in a vector with length one. This correlation is not detected in the abstract interpretation. In order to solve this, first the way the subsystem is transferred into source code was changed. In the initial version of the code, the subsystem was in-lined. Consequently, more than one code segment would need to be annotated. The model was updated such that the calculation of the unit vector is contained in a separate C function. So, the number of statements where an annotation is necessary is reduced. The return value of the C function can be annotated once. To get the range of the return value, a small Gappa script is compiled. The script is based on the fact that in the

mathematical correct case, all components of a unit vector are in the range of $[-1.0,1.0]$. Additionally, the rounding errors introduced by the calculation on the target are considered. The rounding error with the biggest effect is that, as described in chapter 6, the implemented square root has a precision of $\pm 1\,ulp$. The result is the potential range of the result value considering the rounding errors of the floating-point calculations. The range is slightly bigger than $[-1.0,1.0]$. This range was transferred using a known range annotation on each component of the return value of the unit vector function. Still, the fact that the sum of all components should be 1.0, is not contained. But with the software at hand the range limitation of each component was sufficient enough to eliminate false alarms.

A similar approach is also chosen for the calculation of the Bezier curve, which is used for the reference trajectory of the approach in the auto land module [93]. First for all three applicable segments, the Bezier curves for the position and the three derivatives are transferred to Sollya. As a Bezier curve can be reformulated as polynomial, Sollya can be used to determine the maxima and minima of the curve with possible input values in the range of $[0.0,1.0]$.These values are much tighter than the ones calculated by Astrée. During the abstract interpretation, the range from each argument in each calculation step is considered independently from the following and previous steps. This can lead to a huge overestimation for the evaluation of polynomials, as the fact that one of the inputs stays constant is not considered. In order to consider the rounding errors during the evaluation with double precision numbers, Gappa is used. The results are then transferred to the Astrée analysis by a range annotation to the x, y and z comment of the trajectory data. For some parts of the trajectory geometry sub-system, the same approach is chosen. Here, the vectors of the different derivation grade are associated. As some divisions are involved, the issue is that the correlation between the ranges of the dividend and the divisor are normally not part of the abstract interpretation. Here problems especially occur if the analyzed values of both components range from low absolute values to high absolute values. Then the upper bound of range of the result of the division will be determined by the biggest possible value of the divisor, divided by the smallest possible value of the dividend. But in the case at hand, the divisor and the dividend are correlated in a way that small values of the dividend are only combined with small values of the divisor. Therefore, in the abstract interpretation, the result is again overestimated. Here Sollya is used to reduce the result in combination with a range annotation.

With these improvements, a small number of overflows occurred in the auto land system. These can be solved by restricting the distance between the reference trajectory and the aircraft to 100 km and the position of the aircraft in the ATOL coordinate system to the plus / minus the radius of the earth. Indeed, as neither the ATOL system nor bigger parts of it are capsulated in enabled or action subsystems, most calculations are performed all the time, and so this limits might get violated. But the results of the system are only of interest when the ATOL mode is triggered. This is only the case when the aircraft is close to the airport and so also close to the reference trajectory and to the reference point. As a result, the assumption is valid.

Similar to the calculation of the unit vector, the abstract interpretation cannot analyze that in case of a projection of a vector to another vector, the length of the original vector is not increased. This kind of projection is often used in the trajectory generation module. In order to solve this issue, a small stub was created. This stub first calculates the length of the vector, which should be projected by squaring and adding its components and applying the square root. Afterwards, in order to assign a value to each component of the result for each entry, the length is multiplied with an Astrée volatile input [77] in the range $[-1.0,1.0]$. As the calculation

is repeated for each component, it is important that the factor is volatile, that no correlation between the components will be assumed in the analysis. This is important to guarantee the soundness of the analysis. The result achieved by the stub is still an overestimation, but in comparison to the original implementation, the range of the components could be restricted significantly.

Also the calculation of the intersection point of two vectors, which is used in the trajectory generation module several times, leads to too large values in the Astrée analysis. Similar to the other cases, the correlations between the vectors, which are given to the system, are too complex to be part of the analysis. Here the assumption is made that the system ensures that a meaningful result is gained by the values put into the system. The system itself also provides an internal check to show if a solution is possible. Based on this check, sometimes an alternative backup value is used. In order to consider this assumption, in the analysis, after the calculation of the intersection point, the components of the result vector are set to a similar range as the range which was analyzed for the input. In general, this range is $[-38458952, 38458952]$ meter. This equals approximately six times the earth radius, which should be sufficient for all cases where the system is used to produce a meaningful result. In the trajectory generation module, the value of tau is limited in addition by a range annotation. Tau in the context of the trajectory generation represents the length of the transition maneuvers between a straight leg and a circle or different circle segments. It can either be an angle in radian or the actual length of the transition maneuver measured in meters. In some calculations a too big tau leads to an overflow. But from system point of view, as tau only measures the transition, the values will not get such big. Especially as the calculations, which might lead to an overflow, are encapsulated in action subsystems and so only will execute as the aircraft actually performs a transition. Only the cases where tau expresses a length in meters lead to problems. With the system aspects at the beginning of the applicable subsystems, tau was restricted to a length of 10 km, which is sufficient enough for all transitions.

Most of the alarms, which occur are of the category “Overflow in arithmetic” but in the flight planning unit, where the flight-plans are handled, also some “Out of array bounds” violations occur. The reason is that the end of flight-plan is detected based on the fact, that the ID of the next waypoint equals zero. As the flight-plan lists are considered as reliable source no further protection is implemented here. The fact that the index of the current array element is not further increased, if an element with the ID zero is accessed, is not detected in the abstract interpretation. The first difficulty here is, that all flight-plan arrays have a different number of entries. Additionally, the index is updated at a different part of the code than the one where it is indicated that the index should not be updated. In order to compensate this, a range annotation of the index is introduced which limits the index from 0 to 30. This bound correlates to the longest flight-plan, which is stored in the software at hand. The interface, which enables an update of the flight-plans during the runtime, contains checks that the bounds of the list are not violated. But as the flight-plans are a part of the software which is also changed regularly, a limitation of the index accessing the array perhaps would ensure a more robust behavior of the software in case wrong flight-plans are compiled.

With the annotations described above, almost all alarms in the Simulink software could be eliminated. In the library stubs, some of the contained assert annotations [77] still raise an alarm. The aim of the assert annotations is to show that the design range of the libraries are not violated and consequently, the precision annotations presented in this work apply. The alarms are a range violation of the sine and cosine in the call context of the functions:

- tg_DerivBoth
- tg_DerivHor
- tg_DerivVert

In these functions, the movement of the aircraft is extrapolated in order to calculate differential quotients. The problem with the too wide range occurs in cases where the aircraft is close to the poles. In this case, the predicted values of longitude lambda can get high values. Independent of the validity of the assumption, sine and cosine still will result a value which is close to the correct result, but especially for values close to an integer multiple of $\pi/2$ the precision of the result will decrease a bit more than in the proofs of this work. Considering that the prediction is anyway an estimation, this will be acceptable. But as the algorithms are not necessarily designed to work close to the poles, some annotations were added to show that for a nominal operation envelope, the alarm will not occur. The following values were restricted with a range annotation:

- Absolute kinematic speed to $[25m/s, 95m/s]$
- Load factor in the z direction of the kinematic frame $[-10,10]$
- Cosine of the latitude $[0.1736; 1.0]$, this correlates to a maximum absolute latitude of 80 degree

With those restrictions the predicted lambda value, which is the critical input to sine and cosine, can be reduced to a range with an absolute maximum value a bit bigger than π. This value is within the design range of the elementary functions. With these annotations it is possible to show that none of the previously defined design ranges of the elementary math functions is violated.

9 Execution Time Analysis

As introduced in section 2.3, determining the WCET is one of the mandatory tasks for safety critical airborne software. Therefore, this chapter deals with the timing analysis of the complete integrated software. Some aspects of the WCET are already discussed in a previous publication [57], which discusses the WCET analysis of an earlier version of the application software. The publication focuses on how optimization techniques during the code generation and the compilation can improve the WCET. In this thesis this topic is not further discussed. Also, during the elementary function development, some issues regarding timing already have been addressed. These analyses do not cover the influence of the cache, as the configuration is set to always hit [53] for both the data and the instruction cache. In this chapter the behavior of the cache is considered. In addition, it is also investigated how the timing can be influenced in case some parts of the software are locked to the cache.

In [57], the impact of the used compiler and the impact of different settings in the Embedded Coder are already discussed. As the use of a combination of optimization during the source code generation and the compilation leads to a minimal WCET, here this configuration is used. The paper also proposes some best practice for the implementation of certain constructs in the Simulink model. These are sometimes similar to the ones proposed for the runtime error analysis presented in chapter 8. This is due to the fact that both analyses use the method of abstract interpretation. The difference is the applied domains. So for example, aiT does not perform a value analysis of the floating-point values and for the value analysis only the interval domain is used [53]. But aiT for example provides special domains for the cache analysis, which are not needed for the runtime error analysis in Astrée.

One extension to the execution time analyses presented in [57] is that here the cache is of further interest. In the analysis due to many possible states, the content of the caches can only be approximated. As a result, the fact that cache is used introduces uncertainties in the analysis. As one of the most important properties of the WCET analysis is that sound results are produced, uncertainties normally lead to a higher overestimation. This problem is present almost in all hard real-time systems which use complex computer architectures.

In order to deal with this problem, quite many solutions can be used. The first one is to cover this by a precise analysis, which models the content of the cache. With the use of aiT, this aspect is already covered here. Another approach is cache locking. As shown in section 1.2, here many techniques exist. The method can either have the goal to actually reduce the measured execution time or to reduce the overestimation of the WCET analysis. The applied techniques can be divided in different categories. The first is dependent on the kind of cache, which is locked. It is possible to either lock the instruction cache, the data cache or both of them. Another possible category is the frequency of locking the cache. Basically, there are static and dynamic approaches [30–40]. Statically approaches preload the cache and lock the cache once during the initialize phase. Therefore, the memory regions, which are locked, stay the same during program execution. In difference approaches, which use a dynamic locking, the memory regions, which are locked, change during the program execution. In order to achieve this, the routines which execute the locking, need to be called during the program execution.

For the execution time analysis, the model-based part is of interest. For that reason, in this work dynamic cache locking is associated with a change of the locked memory regions during

the execution of the generated step functions. As described in section 3.4, the system at hand executes the tasks without preemption. So, the execution of the model is not interrupted by any function. In order to implement a dynamic cache locking, during the model execution, two possibilities exist. Either the generated code is modified such that the calls to the routines are integrated, or the cache locking is triggered via a certain trap or interrupt. The first variant has the downside that it affects traceability between the model and the code. Additionally, it increases the documentation effort, and so reduces some of the benefits of a model-based development. The second approach can lead to an increased effort during the execution time analysis. Also, as the routines which apply the locking are executed during runtime the design is more curtail than for static locking. The code size and the number of instructions of the cyclic executed code increases. Especially in cases for instruction cache locking, where the routine which locks the cache normally is inhibited from caching, the execution time also can increase by the cache locking. In order to omit this, complex analyses are necessary, which determine where and when to lock the cache considering the potential adverse effects of the locking routines. Such an analysis needs to be executed on the executable object code. But as DO-178C [2] demands that the complete software is documented in requirements, the analysis and the addition of the locking routines afterwards would introduce at least one additional loop in the software development process. Due to these downsides, dynamic cache locking is not considered here.

Another category for the cache locking approaches is the amount of cache which is locked. Basically, there is the difference between approaches where the whole cache is locked and ones which only lock parts of the cache. Locking the whole cache is easier from a verification point as in those case no cache analysis is necessary. But this leads to the fact that there are parts that are never cached, which might decrease the performance. If only parts of the cache shall be locked, this needs to be supported by the hardware. For some CPU, it is possible to lock certain cache lines. For the hardware used here it is only possible to lock whole ways of the cache [15]. Therefore, a partial locking of the cache is available and as the verification tool chain supports a cache analysis it is possible to perform also a partial locking of the cache. As always a complete way must be locked, it is easier to lock contiguous memory regions of at least 32 bytes times 128 sets per way, which equals: 4 KB. Otherwise, much attention must be payed that the memory which shall be locked actually fits into a cache way and as many ways as possible are actually allocated. The locking starts at way zero and afterwards the consecutive ways can be locked. As only way locking is available, the granularity of locking is 4 KB. The limitation of locking only whole ways in a consecutive manner makes dynamic cache locking not very attractive for the hardware at hand.

The goal of the cache locking here is mainly to reduce the analyzed WCET, because for hard real-time systems this is the limiting factor. As stated above, a dynamic locking is not applicable for the system at hand. As the system basically only executes one main task the complete locking is done during system initialization directly after the boot loader. Similar to most of the works presented in the state of the art in section 1.2, here also the parts, which are accessed frequently, are selected as most beneficial for the locking to the cache. As the CPU uses a LRU replacement policy in the actual execution, these parts might anyway be in the cache. Thus, locking such parts might not impact the actual execution time. But due to the uncertainty in the analysis, these parts might be treated as a miss several times and, therefore, have a bigger impact on the analyzed WCET.

One of the downsides of the software at hand is that most of the Simulink subsystems and libraries are in-lined in the source code. The reason therefore is that in order to comply with

the SLCI, all these sub-functions would need to be transferred to model references. But on model level this leads to a quite low performance in simulations, which is a high burden for the agility of the development. Due to the in-lining of the subsystem, the frequently accessed functions in the Simulink part are quite limited.

In advance of performing the cache locking, first the impact of the cache size on the analyzed WCET is determined. Therefore, eighteen analyses were performed. The size of the data or instruction cache was reduced by one way for each analysis. For the other cache, all eight ways were set as available. The memory configuration was not changed and nothing was marked as locked to a cache. The results are shown in Figure 9-1.

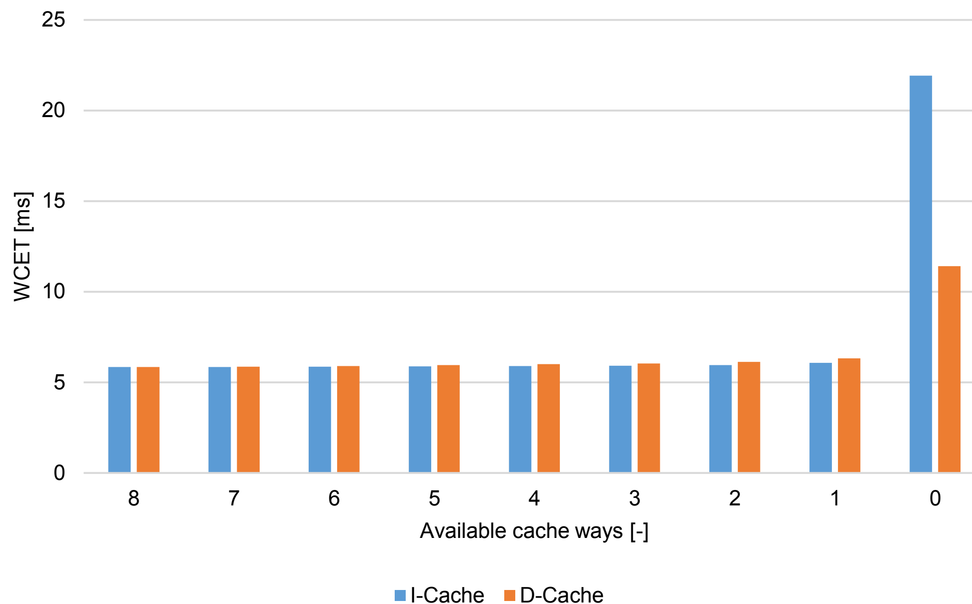

Figure 9-1: Analyzed WCET over Available Cache Ways

Interesting is the fact that the analyzed WCET is almost the same for the cases where at least one cache way is available. For both the data cache and the instruction cache, the WCET monolithically increases with less available cache ways. But the difference between all eight ways available versus only one way available is for the instruction cache about 3.2 % and for the data cache approximately 8.3 %. Only in cases with no cache available at all the WCET increases significantly. Due to the results cache locking is probably quite promising in order to reduce the analyzed WCET. The reason for this behavior is probably that due to the architecture of the code with big functions with many in-lined operations, spatial locality will have the greatest impact on the analyzed WCET. In order to show that the following instructions are already cached, a small cache size is sufficient.

In order to analyze the impact on the WCET, additionally to the baseline two further configurations were analyzed. The configuration of the baseline is similar to the version of [57], where optimizations are applied by the Embedded Coder and the compiler. Here, no cache locking is applied and for both data and instruction cache all eight ways are available in the analysis. The other two variants are described in section 9.1 and 9.2. In difference to the baseline version, first only one way is preloaded and locked. Here, mainly the data and code of the elementary functions is selected for the locking. In a second variant six ways are preloaded and locked. In this case, also frequently used data and code of the model-based developed part is locked.

For all variants next to the WCET analysis, the execution time also has been measured in a hardware in the loop (HIL) setup. Next to the FCC, this setup contains all physical interfaces and a real-time simulation of the aircraft dynamics and sensor dynamics. In order to get an estimate for the maximum measured execution time, each software cycle is measured based on a FCC internal timer. This time is transmitted via CAN and recorded. Based on the recorded data, the maximum time can be evaluated. In order that the recoded data are meaningful, different software modes must be measured. To ensure this for all variants, a complete flight with all different operation modes has been simulated. Although, the test has triggered the different high-level modes, the resulting time still has to be taken with care. Due to some limitations, the time still might not be the highest execution time, which is observable in a closed loop system. Examples for these limitations are: The complete simulation only represents nominal behavior. It does not include non-nominal behavior like temporary sensor failures. All inputs to the software have the expected ranges and rate of changes. In addition to this, due to robust programming it might not be possible to trigger the path leading to the worst-case execution with external inputs only. The fact that the measured time is not the worst-case impacts mainly that the observed overestimation of the WCET has to be taken with care. But the number is still very useful to evaluate the impact of the cache locking on the execution time during nominal operation.

9.1 First Variant

The aim of the first variant is to lock the following to the cache:

- The basic libraries called by the generated code to the instruction cache. The development of those libraries is described in chapter 5, 6, and 7
- The data used by the libraries to the data cache

These were selected as the overall strategy here is to lock often used memory sections to the cache, and next to a few shared utility functions, the libraries are the functions with the most calls on the WCET path. Also, for the selection of the library functions the timing analysis was performed with the cache setting to always hit, the validity of this assumption is actually only true in case these functions are actually preloaded and locked.

In the actual implementation, a few limitations occur. The code size for the necessary library functions are listed in Table 9-1:

<table>
<tr><th>Function</th><th>Code Size [Bytes]</th></tr>
<tr><td>asin</td><td rowspan="2">0x2D8 = 728</td></tr>
<tr><td>acos</td></tr>
<tr><td>atan2</td><td>0x88 = 136</td></tr>
<tr><td>atan</td><td>0x18C = 396</td></tr>
<tr><td>atanf</td><td>0X150 = 336</td></tr>
<tr><td>cos</td><td rowspan="2">0x32C = 812</td></tr>
<tr><td>sin</td></tr>
<tr><td>cosf</td><td rowspan="2">0x2CC = 716</td></tr>
<tr><td>sinf</td></tr>
<tr><td>floor</td><td>0x120 = 288</td></tr>
<tr><td>floorf</td><td>0xA0 = 160</td></tr>
<tr><td>sqrt</td><td>0x74 = 116</td></tr>
<tr><td>pow</td><td>0x6C = 108</td></tr>
<tr><td>tan</td><td>0x12C = 300</td></tr>
</table>

tanf	0x108 = 264
ceil	0x120 = 288
ceilf	0xA0 = 160
roundf	0xC4 = 196
Sum over all	**0x138C = 5004**

Table 9-1: Locked Library Functions

Considering the code sizes in the table above, the sum is bigger than one cache way with a size of 4 KB. As a locking, a partly filled way should be omitted, only the ones with the highest numbers of calls were locked to the library. The functions listed in Table 9-1 before the tanf are locked to the cache. Beginning with the tanf, the following functions are not locked and subjected to the replacement policy. The other issue is regarding the data. Most of the floating-point data in the elementary functions is stored in constants. These are mapped to a certain memory section, which is then preloaded to the cache and locked. For a few characteristic numbers like 1.0 or 0.5, in the code actual numbers are used. In difference to integer numbers, where the number is often directly contained in the instruction, this is not possible for floating-point numbers. So, these data are stored at a certain memory address from where they are loaded to a floating-point register during execution. To ensure preloading and locking this data to the cache, the addresses would be necessary. This would require a mapped constant also for all these basic numbers. This is not done by the actual implementation, as this would decrease the readability of the source code. The linker maps all the “1.0” contained in the source code to one memory address. This is part of the relaxation performed by the linker. By introducing special constants in the elementary functions, the memory footprint of the complete application would increase. Because in the generated code those basic numbers are also directly coded and if they are encoded in a special constant in the elementary function, they would exist twice in the memory. Those double numbers are allocated to the section .rodata.cst8. As the section containing the constants used in the elementary functions is actually much smaller than one cache way, the section .rodata.cst8 is immediately put after these data. The routine executing the preloading and locking loads and locks a whole way. Therefore, additionally to the data stored in the mapped constants, also some of the data stored in the section .rodata.cst8 is loaded and locked to the cache. This does not ensure that all the basic data used in the elementary functions is actually locked in the cache. But as their quantity is not such big the impact should be limited. In case this should be analyzed, the actual addresses must be looked up in the disassembly as the information about this section contained in map file is before the relaxing.

In the following this variant is called 1wayL.

9.2 Second Variant

The aim of the second variant is to lock six ways for both the instruction and data cache. So, the LRU replacement policy is only applied to the two remaining ways. The reason for only selecting six and not seven ways is the limited number of reused statements in the auto generated code.

The locking shall work the same way as before. For the instruction cache, the elements, which should be locked shall be contained in one continuous memory section. This section then can be preloaded and locked to the instruction cache after booting. The selection of the content, which should be locked, is performed after the first WCET analysis of the baseline.

Consequently, after this a remapping to different sections is necessary. As most of the code is automatically generated and the code is also automatically verified against the design model, the changes shall be either done directly in the design model, or as minimal as possible. The decision which elements shall be mapped will change for different software versions and cannot be motivated out of the information available during the design phase. For that reason, the remapping is done after the code generation. This also omits a time consuming regeneration of the code after the first analysis. In order that the changes to the code are minimal, the remapping is restricted to complete functions. To keep the necessary effort to perform the change low, a MATLAB script was compiled which adds the necessary lines to perform the remapping for a list of functions. The script has the limitation that the source code of the function which shall be remapped, is contained in the file which has the same name as the function and the file extension ".c". This is true for all the generated functions, the "shared utils" functions and most of the hand coded functions. The mapping is done by two pragmas, which are added directly at the beginning of the file. From certification point of view, these changes can easily be verified using a "diff" tool and a comparison with the original version of the files.

The content which should be locked to the instruction cache, is based on the number of functions calls on the WCET path. In order to achieve this, a list containing all the routines and the corresponding numbers of calls is generated based on the WCET analysis of the baseline. As aiT lists loops contained in functions here separately, these have to be filtered out, as the mapping is restricted to actual function contained in the C source code. Some of the loops have the highest number of calls in the statistic. But as the first run of a loop is directly followed by the next run in the analysis, the potential cache miss should only occur in the first run. Therefore, locking of loops is not too promising, especially as still 8 KB of the cache are unlocked. To the list of functions then the corresponding size of each entry can be associated. The complete list is sorted based on the number of calls and the functions are selected beginning with a high number of calls, until the size of all selected functions equals the size of all ways which should be locked. The list of the functions is then given to the script explained in the paragraph above. As result the elementary math functions, the shared utilities of Simulink, and some repeatedly used custom modeled functions are locked to the cache.

In order to determine which data shall be locked, the count memory access feature of aiT is used. With the help of this feature, memory areas can be given via annotations for which the read and write accesses are counted. The aim is to lock those areas, which have the most accesses, read and write accesses are added together. In order to solve this problem correctly, normally an access statistic needs to be covered for each 32 byte sized cache line. Afterwards, for each of the 128 sets the six lines with the highest accessed counts need to be selected. This first leads to a tremendous amount of different areas, which would need to be considered. Second, as the cache lines with the highest counts are not necessarily contained in a contiguous memory block, also the effort for the locking routine would be quite high. In the worst case $128 \cdot 6 = 768$ different areas must be preloaded and locked to the data cache. In order to reduce this problem, the mapping is restricted to contiguous memory sections with a start address aligned to a 4 KB address and a size of a whole cache way. For this case, 90 different areas are considered in the analysis. In order that different areas are not merged in the analysis by aiT and the results are actually listed for each area separately, a small gap needs to be inserted between these areas. So instead of defining areas, which are exactly the size of an entire cache way, the 4 KB are reduced by 16 bytes. Assuming that the accesses are distributed equally over one block this leads to an error of 0.4 %. But with the assumption

of equal distribution, all areas have the same relative error. With the result that the comparison between them is not influenced. The following 6 areas have the highest access values and are selected for locking:

- Two parts in the stack with a gap of 4 KB in between
- The section where the data of the elementary functions are stored, which is also locked in the first variant
- A part of the COMMON section where model states are stored

As a result, in the locking routine there are four loops which lock the selected areas. In case a source code change shall be omitted, a generic routine can be included, which supports the locking of 6 different 4 KB sections. Then, after the analysis, the start addresses of the sections can be configured via a configurable parameter. In the context of certification, this would be the favorite approach. Such an implementation has the benefit that the design and implementation of the locking routine can be executed during the normal development workflow. Regarding the certification considerations, for the parameters it would be beneficial if those are merged with the executable object code as in this case no additional certification effort would be necessary [2].

In the following this variant is called 6wayL

9.3 Results

In this section, the results of the HIL simulation and the WCET analysis are presented. The numbers of the measurements and analyses are presented in Table 9-2:

Variant	Mean of measured execution time $[ms]$	Standard deviation of measured execution time $[\mu s]$	Max measured execution time $[ms]$	Analyzed WCET $[ms]$	Observed Overestimation $[\%]$
Baseline	2.542	95.6	2.860	5.800	102.8
1wayL	2.528	96.1	2.843	5.014	76.4
6wayL	2.400	98.2	2.714	4.624	70.4

Table 9-2: Analyzed WCET and Measured Time

The observed overestimation in Table 9-2 is defined as:

$$\frac{Analyzed\ WCET}{Max\ measured\ time} - 1 \tag{9-1}$$

As described at the beginning of this chapter, the simulation covers a nominal case and so the number needs to be handled with care. But the comparison of the overestimation between the three cases clearly shows that the precision in the cases where the cache is locked, is increased significantly as the overestimation decreases. In comparison with other publications where measurements of the execution time are compared with an analyzed variant [83, 84], especially under the aspects of the limitations in the HIL simulation, the numbers seem quite tight.

The test sequence of the HIL test takes a little bit more than fifteen minutes. So for each test 100,000 samples of the execution time measurement are captured. A distribution of the execution times is shown in Figure 9-2 for the different variants.

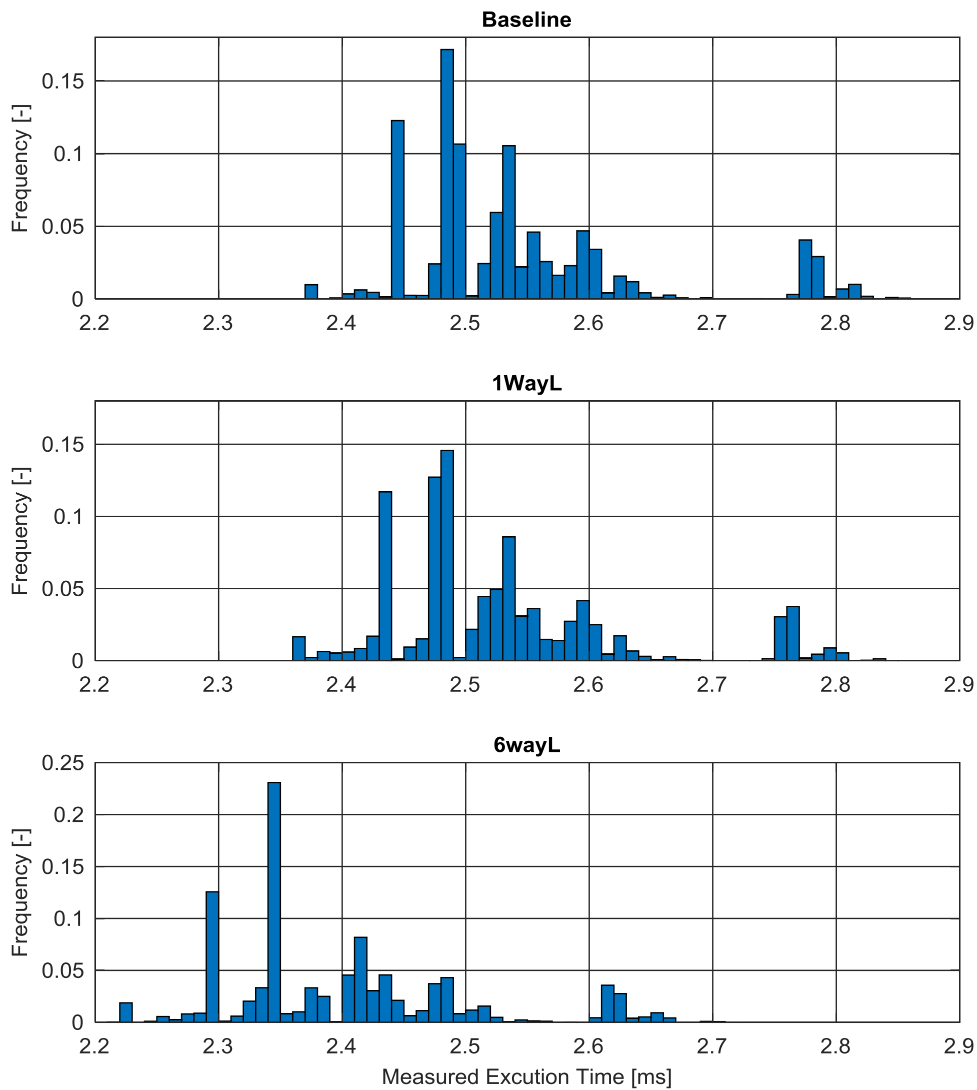

Figure 9-2: Execution Time Distribution

In Table 9-3, the improvement in the analyzed WCET is shown in more detail.

Variant	Absolute WCET Improvement WCET(Baseline) - WCET [ms]	Relative WCET Improvement 1- WCET/WCET(Baseline) [%]
1wayL	0.786	13.6
6wayL	1.176	20.3

Table 9-3: Improvements Achieved via Cache Locking

Interesting is the fact, that locking the first way is quite effective. The absolute improvement by the variant 1wayL is approximately twice as high as the further improvement, which can be reached by locking the next five ways. The benefit of the variant 1wayL is that there all locked functions have at least 12 calls on the WCET path. In contrast to the second variant, where also functions with only 3 calls on the WCET path are locked to the cache. Therefore, the benefit is lower. Additionally, in the second variant also most of the table lookup utilities are

locked to the cache. As these mainly consist out of loops, the percentage of cache hits is there also high in the analysis of the baseline version.

Next to the primary goal to reduce the analyzed WCET, also the measured times are improved. Compared to the baseline, both the average and maximal measured time can be decreased by the application of cache locking. So the actual performance during normal operation is increased, too. Here the benefit of the variant 6wayL is bigger than six times the benefit of the variant 1wayL. The absolute benefit regarding the maximum measured execution time for the variant 6wayL is approximately 8.6 times the benefit of variant 1wayL, which is much higher than the factor 1.5 in the analysis. The factor for the average measured time, which is approximately 10.2, is even higher.

Based on the results, higher improvements of the WCET can probably be achieved, if the generated code would contain more independent functions, which are reused several times. But due to the downsides in the simulation performance, this currently is not attractive in a process, which is strongly driven by analysis base on model simulations on desktop computers. Regarding the WCET of a source code with more separate functions, also the additional instructions through the function prolog and epilog have to be considered. So, in case of a too granular software, this effect might cancel out the effect of cache locking. This is here not further analyzed as the execution frequency of the flight control algorithm is 100 Hz and consequently the analyzed WCET is within the expected bound. In case of the version 6wayL, the margin to the bound of 60 % CPU utilization, requested by [95], is also big enough to account the additional execution time of the manual written software, which is not considered in this thesis.

A comparison of the achieved improvement with the results of the publications presented in section 1.2 is difficult. First of all, most of the referenced papers address applications, which are smaller and also work with smaller caches. The approaches mostly are applied to some benchmark applications and not to real application software. This is interesting, especially as due to the model-based development approach, the application here has kind of special software architecture. As shown above, this circumstance might limit the benefit of cache locking. Most of the referenced work also executes their experiments on special CPU simulations, which only take caching into consideration and not the full complexity of a real CPU. But nevertheless, some comparison can be drawn. First of all, the presented approach here is quite simple, as no iterative execution of the WCET analysis is needed, as for example in [40]. Still, with the approach it was possible to reduce both the actual measured time and the analyzed time. This is definitely a benefit compared to approaches, where the performance was degraded in comparison to a system with the nominal cache replacement policy [31]. As the proposed approach is limited to a single iteration, the impact of cache locking on the WCET path cannot be considered. This impact is considered by [40], but with the burden of an iterative execution of the WCET analysis. In order to omit the iterative process, [37] only executed mandatory parts of the WCET analysis. This was not possible here as the commercial tool aiT is a closed solution, which could not be modified accordingly. Considering these limitations, the achieved improvement is still remarkable and in the same range as the average results of other works like [33, 40]. But the improvement is lower as for example in the work [35] and the best results in [40].

10 Conclusion and Future Work

This chapter presents the achievements of this thesis and also discusses in which directions potentially further developments can go.

10.1 Conclusion

Goal of this thesis is to develop elementary math functions for modern flight management and flight control software. In contrast to different approaches where the development of the elementary math functions are considered, independent from a use case, here the special requirements due to the specific environment are considered. These special requirements include:

- Specific design range for the input to the elementary math functions
- Execution time based selection of the algorithm
- Integration in the overall development process
- Formal proven precision of the implementation
- Verification of the input range with the use of a formal runtime error analysis

The thesis shows that for a safe embedded system, like the software at hand, for many of the needed elementary math functions the input range can be limited to small intervals. The limitation can be motivated by functional considerations and with the integrated software, it is possible to verify that the chosen design range is not violated in the actual use case. This direct coupling of the design of elementary function with the runtime error analysis is an extension to the state of the art. The thesis also shows that in case the elementary functions are adapted to the specific use, parts of the algorithm can be eased in a way that a lower execution time can be reached. An example for this is the application of a simple range reduction algorithm for the sine and the cosine, or the limitation of the power function to integer exponents.

As for controlled systems, the latency of the control loop is one of the limiting factors during the development, the execution time analysis is an integral process of the proposed workflow. Already during the development of the math functions, WCET analyses were executed in order to select the algorithms based on a combination of WCET and reached precision. In order to get a high precision and low execution time, the Horner scheme of the polynomials is used in conjunction with the use of the FMA instruction. This approach was chosen in order to keep a high portability to different targets. In case the execution time needs to be further improved, the execution order of the polynomial also can be further adapted in order to comply with parallelization support of the hardware. In [5], some basic proposal can be found. But due to the intermediate load instructions, which fetch the coefficients from the memory, the proposed scheme did not bring a benefit for the analyzed WCET of a small example for the target at hand. In order to further reduce the computational effort, advanced methods were chosen to deduct the approximation polynomials. These methods minimize the desired approximation error over the complete design range, so that the magnitude of the error is independent from the input. In addition to the feature that the error is kept in a constant magnitude over the complete input range, the chosen approach also considers the limitations of floating-point numbers. By applying these methods, the grade of the approximation polynomial can be kept low, which leads to a reduced computation time.

To verify the precision reached by the approximation functions, some works only consider the approximation error of the elementary functions. This thesis not only considers the approximation error, but also the rounding errors made during execution on the target are taken into account. This is done using formal proofs, which are compiled with the help of the tool Gappa. Gappa can automate many process steps during the creation of the proof. This automation enables a fast process and it protects the introduction of sporadic errors, which can occur in handwritten proofs.

In the last step of the thesis, it is shown how the WCET and the actual execution time of the software can be improved in a combination of partial cache locking and static analysis. As the developed math functions are quite frequently executed, the topic of cache locking is related to the development of such functions in a complete system. In addition to that, the cache locking is also used to show the validity of the assumptions of the intermediate WCET analysis during the function development.

10.2 Future Work

The model integration of specially developed math functions is addressed in this thesis from an architectural perspective and executed only for small isolated examples. So in the future, more effort should be spent investigating the benefits of the different variants via code replacement library or legacy code. The actual integration in the complete model can be executed to analyze the impact on the different aspects. Examples for such aspects are: the system linearization during the system development, the simulation in the desktop environment, the code generation and comparison of the desktop results with the results on the target execution in a PIL environment. In case of the legacy code variant also the relevant non-functional attributes can be investigated and how they can be added to the generated blocks.

As already stated above, one further point of improvement is to work on the parallelization support for the proposed algorithm. As the target at hand has a pipeline in the floating-point unit [15], parallel execution of different floating-point instructions is possible. An analysis of the pipeline could show if the implementation at hand already utilizes the pipeline in an optimal way. If this is not the case, it could be analyzed if refactoring the implementation leads to a better utilization of the pipeline and a lower WCET. In case that a hardware offers more parallelization features, like two parallel FPUs or full vector calculation support, this topic becomes more important.

It was possible to show some improvements in the execution time by applying cache locking. But the achieved improvements are not as high as in some of the previous publications in this field. Beside some other aspects the architecture of the generated code might lead to a limited improvement. So, in future work the model could be split, also automatically, into smaller granular functions, which then are also called more frequently. Furthermore, it could be analyzed if in this case, the cache locking is more effective. Also, the proposed algorithms to select the objects for cache locking can be compared with the state-of-the-art algorithms applied to the current software. Currently the proposed, and also most of the state-of-the-art algorithms for cache locking, rely on the executable object code. But as here the design model is already a formal executable representation of the software, it could be possible to develop some cache locking algorithms utilizing the model information. The benefit would be that the cache locking objects would be already known a priori to the first executable object code, and

an additional iteration step could be omitted. The challenge of such an approach is that between the model and the executable, there are two process steps. First, there is the translation of the model to source code by the Embedded Coder. Afterwards, the second step is the compilation and linking of the actual target binary. Both of these steps perform optimizations, so they have an impact on the output and must be considered in case the parts, which shall be locked, shall be selected a priori. Additionally, to include the effect of these process steps, some parts of the target need to be considered. These parts should also be coverable by a model. For the cache locking algorithm, it could be further analyzed if there is the possibility to incorporate the locations of cache hits and misses of the static WCET analysis, as it would not make sense to lock parts, which are anyway detected as cache hits.

During this thesis, the focus was on implementing two precision variants for each elementary math function: One which achieves a precision of $\pm 1ulp$ for single precision number, and the second one with a precision as high as possible with pure double precision arithmetic. The experiments show that these precisions are high enough that the target software behaves as expected. In case the precision is reduced, at some points an unexpected behavior is observed. In future work the actual necessary precision could be determined. Such considerations should take into account the measurement resolution, which is often much lower than double precision. The accuracy of the actuation path shall be considered too. With these limitations, there is probably a limit at which an increased precision of the calculation of the FCC does not bring a benefit for the overall solution. In order to determine such limits, either formal analyses based on the model or based on the source code could be developed. In case such an analysis reviles that a lot of different precisions need to be provided, a tradeoff between the memory consumption and providing functions with adapted precision must be found.

As the development approach in [57] and in this thesis use a formal verified compiler, no errors should be introduced in the compilation of the source code. In addition, this thesis enables that a model can be developed without relying on external components, like the math.h library. In case the developed elementary functions are provided via the legacy code option, simulation and execution on the target should become equal to the LSB. In further work, it can be analyzed how this fact can be utilized in the context of a certification project. In case it can be shown that all the steps between the model and the simulation and between the model and the executable object code are correct, some target testing could potentially be replaced by simulation. This has the benefit, that simulation is normally faster than testing on the real hardware and also less hardware would be necessary in this case. As the simulation mainly represents the functionality and not non-functional topics, like memory and time resources, for sure not all tests could be substituted, but especially for verifying the functional behavior this could be a promising approach. One topic of investigation could be if using the formally verified compiler, also on the development host brings a benefit for the certification.

Bibliography

[1] *COMPILER-SUPPLIED LIBRARIES*, CAST-21, 2004.
[2] *Software Considerations in Airborne Systems and Equipment Certification*, DO-178, 2011.
[3] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754, 2008.
[4] *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std 754, 1985.
[5] J.-M. Muller, *Elementary Functions: Algorithms and Implementation,* 3rd ed. Boston, MA: Birkhäuser, 2016.
[6] J. Schumann and S.-A. Schneider, “Automated Testcase Generation for Numerical Support Functions in Embedded Systems,” in *NASA Formal Methods Symposium*, Houston, TX, USA, 2014.
[7] M. Beemster, “Learning from Math Library Testing for C,” Solid Sands, Mar. 2018.
[8] *MARK 33 DIGITAL INFORMATION TRANSFER SYSTEM (DITS)*, ARINC SPECIFICATION 429, 2014.
[9] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand, “Obtaining Worst-Case Execution Time Bounds on Modern Microprocessors,” in *Embedded World Conference*, Nürnberg, Germany, 2018.
[10] C. Vinschen and J. Johnston, *The Newlib Homepage.* [Online] Available: https://sourceware.org/newlib/. Accessed on: November 2018.
[11] J. Hart, *Computer approximations*. Huntington N.Y.: R.E. Krieger Pub. Co, 1978.
[12] J. Detrey and F. de Dinechin, “Floating-Point Trigonometric Functions for FPGAs,” in *International Conference on Field-Programmable Logic and Applications (FPL)*, Amsterdam, Netherlands, 2007, pp. 29–34.
[13] F. de Dinechin, M. Joldes, and B. Pasca, “Automatic generation of polynomial-based hardware architectures for function evaluation,” in *21st IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, Rennes, France, 2010, pp. 216–222.
[14] P. Tang, “Table-lookup algorithms for elementary functions and their error analysis,” in *Computer arithmetic: 10th Symposium : Papers*, Grenoble, France, 1991, pp. 232–236.
[15] I. Freescale Semiconductor, *e300 Power Architecture™ Core Family Reference Manual*, 2007.
[16] NXP Semiconductors, *e200z4 Power Architecture™ Core Reference Manual*. October 2009.
[17] Infineon Technologies AG, *TriCore V 1.6 Core Architecture User Manual (Volume 1)*, 2015.
[18] *Design Assurance Guidance for Airborne Electronic Hardware*, DO-254, 2000.
[19] W. Lee, R. Sharma, and A. Aiken, “On automatically proving the correctness of math.h implementations,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, 2018.
[20] J. Harrison, “Formal Verification of Floating Point Trigonometric Functions,” in *Formal methods in computer-aided design (FMCAD)*, Austin, TX, USA, 2000, pp. 254–270.
[21] Texas Instruments, “AM572x Sitara™ Processors Silicon Revision 2.0,” Oct. 2018.
[22] T. Lorenser, “The DSP capabilities of ARM® Cortex®-M4 and Cortex-M7 Processors: DSP feature set and benchmarks,” ARM, Nov. 2016.

[23] N. Shibata, “Efficient evaluation methods of elementary functions suitable for SIMD computation,” *Computer Science - Research and Development*, vol. 25, no. 1-2, pp. 25–32, 2010.

[24] Jingyan JOURDAN-LU, “Custom floating-point arithmetic for integer processors: algorithms, implementation, and selection,” Dissertation, University of Lyon, Lyon, 2012.

[25] R. Kirner, M. Grössing, and P. Puschner, “Comparing WCET and Resource Demands of Trigonometric Functions Implemented as Iterative Calculations vs. Table-Lookup,” in *6th International Workshop on Worst-Case Execution Time Analysis*, Dresden, Germany, 2006.

[26] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic optimization of floating-point programs with tunable precision,” in *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Edinburgh, United Kingdom, 2014, pp. 53–64.

[27] C. Daramy-Loirat *et al.,* “CR-LIBM A library of correctly rounded elementary functions in double-precision.,” Dec. 2006.

[28] G. Melquiond, *User’s Guide for Gappa*, 2016.

[29] D. Delmas *et al.,* “Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software,” in *International Workshop on Formal Methods for Industrial Critical Systems*, Eindhoven, Netherlands, 2009.

[30] A. Arnaud and I. Puaut, “Dynamic instruction cache locking in hard real-time systems,” in *REAL-TIME AND NETWORK SYSTEMS*, Poitiers, France, 2006.

[31] X. Vera, B. Lisper, and J. Xue, “Data cache locking for higher program predictability,” in *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, San Diego, CA, USA, 2003, p. 272.

[32] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals, “Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems,” *Journal of Systems Architecture*, vol. 57, no. 7, pp. 695–706, 2011.

[33] H. Ding, Y. Liang, and T. Mitra, “WCET-Centric dynamic instruction cache locking,” in *Design, Automation & Test in Europe*, Dresden, Germany, 2014, pp. 1–6.

[34] V. Suhendra and T. Mitra, “Exploring locking & partitioning for predictable shared caches on multi-cores,” in *45th Design Automation Conference (DAC)*, Anaheim, CA, USA, 2008, pp. 300–303.

[35] R. Mancuso *et al.,* “Real-time cache management framework for multi-core architectures,” in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Philadelphia, PA, USA, 2013, pp. 45–54.

[36] I. Puaut and D. Decotigny, “Low-complexity algorithms for static cache locking in multitasking hard real-time systems,” in *23rd IEEE real-time systems symposium: (RTSS 2002)*, Austin, TX, USA, 2002, pp. 114–123.

[37] H. Falk, S. Plazar, and H. Theiling, “Compile-time decided instruction cache locking using worst-case execution paths,” in *5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Salzburg, Austria, 2007, pp. 143–148.

[38] T. Liu, M. Li, and C. J. Xue, “Minimizing WCET for Real-Time Embedded Systems via Static Instruction Cache Locking,” in *15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, CA, USA, 2009, pp. 35–44.

[39] M. Campoy, A. Perles Ivars, and J. V. Busquets Mataix, “Static Use of Locking Caches in Multitask Preemptive Real-Time Systems,” in *Real-Time Embedded System Workshop*, London, England, Dec. 2001.

[40] H. Ding, Y. Liang, and T. Mitra, “WCET-centric partial instruction cache locking,” in *49th Design Automation Conference (DAC)*, San Francisco, CA, USA, 2012, pp. 412–420.

[41] NXP Semiconductors, *MPC5777C Reference Manual*, 2016.
[42] ARM, *Arm® Cortex® -R52 Processor: Technical Reference Manual*, 2017.
[43] Intel Corporation, “How to detect New Instruction support in the 4th generation Intel® Core™ processor family,” 2013.
[44] R. Wilhelm *et al.,* “The Worst-case Execution Time Problem Overview of Methods and Survey of Tools,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, 36:1–36:53, 2008.
[45] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.
[46] W. Kahan, “Why do we need a floating-point arithmetic standard?,” University of California at Berkeley, Feb. 1981.
[47] *ANSI C 1999*, ISO/IEC 9899, 1999.
[48] D. R. Lutz, “Fused Multiply-Add Microarchitecture Comprising Separate Early-Normalizing Multiply and Add Pipelines,” in *20th IEEE Symposium on Computer Arithmetic (ARITH)*, Tuebingen, Germany, 2011, pp. 123–128.
[49] J. M. Muller, *Handbook of floating-point arithmetic*. Boston, Mass.: Birkhäuser, 2010.
[50] P. H. Sterbenz, *Floating-Point Computation*. Englewood Cliffs, NJ: Prentice-Hall, 1974.
[51] K. Wüst, *Mikroprozessortechnik: Grundlagen, Architekturen, Schaltungstechnik und Betrieb von Mikroprozessoren und Mikrocontrollern,* 4th ed. Wiesbaden: Vieweg+Teubner Verlag / Springer Fachmedien Wiesbaden, Wiesbaden, 2011.
[52] I. Freescale Semiconductor, *MPC8349EA PowerQUICC II Pro Integrated Host Processor Family Reference Manual*, 2006.
[53] AbsInt Angewandte Informatik GmbH, *a3 for e300 User Documentation*, 2017.
[54] T. Flik, *Mikroprozessortechnik und Rechnerstrukturen,* 7th ed. Berlin, Heidelberg, Germany: Springer-Verlag, 2005.
[55] P. G. Hamel, *In-flight simulators and fly-by-wire/light demonstrators: A historical account of international aeronautical research*. New York NY: Springer-Verlag, 2017.
[56] H. Theuerkauf, “Interface Control Document of Flight Control Computer FCC 9241,” Aircraft Electronic Engineering GmbH, Jun. 2016.
[57] K. Nürnberger, M. Hochstrasser, and F. Holzapfel, “Execution time analysis and optimisation techniques in the model-based development of a flight control software,” *IET Cyber-Physical Systems: Theory & Applications*, vol. 2, no. 2, pp. 57–64, 2017.
[58] MicroSys GmbH, “User’s Manual MPX8349 Rev. 3,” MicroSys GmbH.
[59] C. Krause and F. Holzapfel, “System Automation of a DA42 General Aviation Aircraft,” in *Aviation Technology, Integration, and Operations Conference*, Atlanta, Georgia, USA, 2018.
[60] V. Schneider, “Trajectory Generation for Integrated Flight Guidance,” Dissertation, Technische Universität München, Munich, 2018.
[61] D. M. Gierszewski, V. Schneider, P. J. Lauffs, L. Peter, and F. Holzapfel, “Clothoid-Augmented Online Trajectory Generation for Radius to Fix Turns,” in *15th IFAC Symposium on Control in Transportation Systems*, Savona, Italy, 2018, pp. 174–179.
[62] N. C. Mumm and F. Holzapfel, “Development of an Automatic Landing System for Diamond DA 42 aircraft utilizing a Load Factor Inner Loop Command System,” in *CEAS EuroGNC*, Warsaw, Poland, 2017.
[63] A. W. Zollitsch *et al.,* “Automatic takeoff of a general aviation research aircraft,” in *Asian Control Conference Gold Coast, Australia*, Gold Coast, QLD, Australia, 2017, pp. 1683–1688.
[64] E. Karlsson *et al.,* “Development of an Automatic Flight Path Controller for a DA42 General Aviation Aircraft,” in *CEAS EuroGNC*, Warsaw, Poland, 2017.

[65] S. P. Schatz, “Development and Flight-Testing of a Trajectory Controller Employing Full Nonlinear Kinematics,” Dissertation, Technische Universität München, Munich, 2018.

[66] S. P. Schatz and F. Holzapfel, “Nonlinear Modular 3D Trajectory Control of a General Aviation Aircraft,” in *CEAS EuroGNC*, Warsaw, Poland, 2017.

[67] M. Hochstrasser *et al.,* “Aspects of a Consistent Modeling Environment for DO-331 Design Model Development of Flight Control Algorithms,” in *CEAS EuroGNC*, Warsaw, Poland, 2017.

[68] A. W. Zollitsch, S. P. Schatz, N. C. Mumm, and F. Holzapfel, “Model-in-the-Loop Simulation of Experimental Flight Control Software,” in *AIAA Modeling and Simulation Technologies Conference*, Kissimmee, Florida, USA, 2018.

[69] MathWorks, *Embedded Coder Users's Guide*, 2016.

[70] X. Leroy, “Formal verification of a realistic compiler,” *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[71] X. Leroy, *The CompCert C verified compiler: Documentation and user’s manual,* 2nd ed., 2016.

[72] S. Chevillard, M. Joldeş, and C. Lauter, “Sollya: An Environment for the Development of Numerical Codes,” in *Mathematical Software - ICMS*, Kobe, Japan, 2010, pp. 28–31.

[73] N. Brisebarre and S. Chevillard, “Efficient polynomial L-approximations,” in *18th IEEE Symposium on Computer Arithmetic (ARITH)*, Montpellier, France, 2007, pp. 169–176.

[74] S. Chevillard, J. Harrison, M. Joldeş, and C. Lauter, “Efficient and accurate computation of upper bounds of approximation errors,” *Theoretical Computer Science*, vol. 412, no. 16, pp. 1523–1543, 2011.

[75] F. de Dinechin, C. Lauter, and G. Melquiond, “Certifying the Floating-Point Implementation of an Elementary Function Using Gappa,” *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 242–253, 2011.

[76] M. Hornauer and F. Holzapfel, “Modellbasiertes Testen für CS-23 Avionik und UAV Anwendungen,” in *DGLR Workshop: Verifikation in der modellbasierten Software-Entwicklung*, Garching; Germany, 2011.

[77] AbsInt Angewandte Informatik GmbH, *a3 for C User Documentation*, 2017.

[78] *Model-Based Development and Verification Supplement to DO-178C and DO-278A*, DO-331, 2011.

[79] *Formal Methods Supplement to DO-178C and DO-278A*, DO-333, 2011.

[80] M. Hochstrasser, C. Krause, V. Schneider, and F. Holzapfel, “Model-based Implementation of an Onboard STANAG 4586 Vehicle Specific Module for an Air Vehicle,” in *AIAA SciTech Forum: AIAA Modeling and Simulation Technologies Conference*, Grapevine, Texas, USA, 2017.

[81] *Software Tool Qualification Considerations*, DO-330, 2011.

[82] The Coq Development Team, *The Coq Reference Manual,* 8th ed., 2018.

[83] P. Baufreton and R. Heckmann, “Reliable and Precise WCET and Stack Size Determination for a Real-life Embedded Application,” in *ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation*, Poitiers, France, 2007, pp. 41–48.

[84] J. Souyris *et al.,* “Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation,” in *5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, Palma de Mallorca, Spain, 2005, pp. 21–24.

[85] J. Bertrane *et al.,* “Static Analysis and Verification of Aerospace Software by Abstract Interpretation,” in *AIAA Infotech@Aerospace 2010*, Atlanta, Georgia, USA, 2010.

[86] P. Cousot, “Formal Verification by Abstract Interpretation,” in *4th International Symposium NASA formal methods (NFM)*, Norfolk, VA, USA, 2012, pp. 3–7.

[87] R.-C. Li, Boldo Sylvie, and M. Daumas, “Theorems on efficient argument reductions,” in *16th IEEE symposium on computer arithmetic*, Santiago de Compostela, Spain, 2003, pp. 129–136.

[88] M. Abramowitz and Stegun Trene A., *Handbook of Mathematical Functions: with Formulars, Graphs, and Mathematical Tables,* 10th ed. Washington, D.C.: U.S. Government Printing Office, 1972.

[89] NXP Semiconductors, *Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture*, 2005.

[90] MathWorks, *Simulink Reference*, 2016.

[91] MathWorks, *Simulink Code Inspector Reference*, 2016.

[92] MathWorks, *DSP System Toolboy Reference*, 2016.

[93] D. Tromba, L. Munteanu, V. Schneider, and F. Holzapfel, “Approach trajectory generation using Bezier curves,” in *IEEE Intenational Conference on Aerospace Electronics and Remote Sensing*, Bali, Indonesia, 2015.

[94] MathWorks, *Polyspace Code Prover Reference*, 2018.

[95] *AEROSPACE RECOMMENDED PRACTICE Aerospace - Vehicle Management Systems - Flight Control Design, Installation and Test of, Military Unmanned Aircraft, Specification Guide For*, ARP94910, 2012.

Appendix

A Polynomial Coefficients

Here the polynomial coefficients of the used by the selected functions are listed. The number indicates the degree of the corresponding exponent.

A.1 Cosine

Relative Approximation Error 3.2614e8
s0 = 0.99999996738614616020868197665549814701080322265625;
s2 = -0.49999842433970742572313383789150975644588470458984;
s4 = 4.1654419553239729889781983729335479438304901123047e-2;
s6 = -1.3579403997328709074315922222808694641571491956711e-3;

Relative Approximation Error 1.1103e-16
s0 = 0.99999999999999998889776975374843459576368331909179 7;
s2 = -0.49999999999999891198143586734659038484096527099609 4;
s4 = 4.1666666666423747555914047779879183508455753326416e-2;
s6 = -1.3888888867084076689240035307193465996533632278442 4e-3;
s8 = 2.4801577714838995557689732551764905110758263617754e-5;
s10 = -2.7555122689746407411515247837252573503974417690188e-7;
s12 = 2.0624728115079714729865728833468202507717137450527e-9;

A.2 Sine

Relative Approximation Error 3.2413-9
s1 = 0.99999999676179707197576362887048162519931793212 89;
s3 = -0.16666650224234560995206777533894637599587440490722 7;
s5 = 8.3320164528557730948543635918213112745434045791626e-3;
s7 = -1.9501821991924086726645504175081669018254615366459e-4;

Relative Approximation Error 3.7828e-18
s1 = 1.0;
s3 = -0.16666666666666662965923251249478198587894439697265 6;
s5 = 8.3333333333219204719366501876720576547086238861084e-3;
s7 = -1.9841269829461250580500408791095878768828697502613e-4;
s9 = 2.7557313583628474970121501747000181126168172340 84e-6;
s11 = -2.5050742590829393772440951177089829560173939171363 6e-8;
s13 = 1.5895962194970944904556161340712780102868428855345e-10;

A.3 Tangent

Offsets
o0 = 0.0;
o1 = 0.19891236737965800607241817488102242350578308105469;
o2 = 0.41421356237309503445231939622317440807819366455078;
o3 = 0.66817863791929887895548745291307568550109863281 25;
o4 = 0.99999999999999998889776975374843459576368331909179 7;

o5 = 1.49660576266548894786012624535942450165748596191406;
o6 = 2.41421356237309492343001693370752036571502685546875;
o7 = 5.0273394921258462986202175670769065618515014648437;
o8 = 16331239353195370;

Relative Approximation Error 1.6549e-9
s1 = 1.0000000015194612235092108676326461136341094970703;
s3 = 0.33333049683824900188966466885176487267017364501953;
s5 = 0.13411706332367132055161107473395531997084617614746;

Relative Approximation Error 1.6533e-18
s1 = 1.0;
s3 = 0.33333333333333410308796374010853469371795654296875;
s5 = 0.133333333322169678147872673434903845191001892089844;
s7 = 5.3968259256325548656185731033474439755082130432129e-2;
s9 = 2.18683929411041810819682496003224514424800872802734e-2;
s11 = 8.966297729047681408509617995150620117783546447754e-3;

A.4 Arcus Tangent

Interval Limits
X_i:
X_1 = 0.26794919243112269580109341404750011861324310302734;
X_2 = 1.0;
X_3 = 3.7320508075688771931766041234368458390235900878906;

x_i:
x_1 = 0.57735026918962573105886804114561527967453002929687;
x_2 = 1.73205080756887719317660412343684583902359008789062;

atan(x_i):
atan(x_1) = 0.52359877559829892668119555310113355517387390136719;
atan(x_2) = 1.047197551196597853362391106202226711034774780273437;
atan(x_3) = 1.5707963267948965579989817342720925807952880859375;

Relative Approximation Error 2.0246e-8
s1 = 0.99999997977266208426527782648918218910694122314453;
s3 = -0.33332423445865277944477611526963301002979278564453;
s5 = 0. 19935727066940181484433480818552197888493537902832;
s7 = -0.12813335126290517629321641379647189751267433166504;

Relative Approximation Error 1.6938e-17
s1 = 1.0;
s3 = -0.333333333333330933401228435286611784249544143676758;
s5 = 0.199999999989247084775101370723859872668981552124023;
s7 = -0.14285714119902134799922066576982615515589714050293;
s9 = 0.111110986901310909713558316980197560042142868041992;
s11 = -9.090391985939952346207348909956635907292366027832e-2;
s13 = 7.67968112122032336985810729856893885880708694458e-2;
s15 = -6.485678591510658785868059794665896333754062652588e-2;
s17 = 4.4470912757197278430965070583624765276908874511719e-2;

A.5 Arcus Sine

Interval Limits

1.4901161193847656250e-8
0.38213556644701845987910360236128326505422592163086
0.63167340493143198099090795949450694024562835693360
0.78076933221223532299859471095260232686996459960937
0.86823995276712906399296798554132692515850067138672

Offsets

-6.5790252277528851308929707851689883667028707998173e-24
0.39210614867989368459433308089501224458217620849609,
0.68370989696567463500542771726031787693500518798828,
0.89589615914923648176682036137208342552185058593750

Coefficients Interval 1

s1 = 1.0000000000000002220446049250313080847263336181640б;
s2 = -2.1567647881195321039080605397289541017859758331720бe-13;
s3 = 0.16666666670622676260649086543708108365535736083984;
s4 = -2.9208037775397944286373412873473315309880149470700e-9;
s5 = 7.5000113535655513197220045640278840437531471252441e-2;
s6 = -2.6593207380881241923991677167382619018098921515047бe-6;
s7 = 4.4683429599529811415692392984055913984775543212890e-2;
s8 = -4.2265549353926959051316880611182114080293104052544e-4;
s9 = 3.3472198762578862785144195868269889615476131439209e-2;
s10 = -1.6060641929034672609200029569365142378956079483032e-2;
s11 = 8.1640941999453972566236359398317290470004081726074e-2;
s12 = -0.15265151606238078207944397490791743621230125427246;
s13 = 0.28051981750955340411124439015111420303583145141602;
s14 = -0.27706068807021161370940376400540117174386978149414;
s15 = 0.15310798111738987969943082134705036878585815429687;

Coefficients Interval 2

s1 = 1.0821266192599376942951039382023736834526062011718;
s2 = 0.24211500531392451951440136781457113102078437805176;
s3 = 0.31953630901194651769969823362771421670913696289062;
s4 = 0.27323799100088974611821868165861815214157104492187;
s5 = 0.33955757055130764143768828944303095340728759765625;
s6 = 0.39848222601355948180312793738266918808221817016602;
s7 = 0.51881383758873256262234008318046107888221740722656;
s8 = 0.64600767399750202990560410398757085204124450683594;
s9 = 1.2511306465545157795560271551948972046375274658203;
s10 = -1.4058454828077073806014141155174002051353454589843;
s11 = 16.811858814973049902619095519185066223144531250;
s12 = -56.757752339227479865257919300347566604614257812;
s13 = 159.69665463558288820422603748738765716552734375;
s14 = -246.20940098502921955514466390013694763183593750;
s15 = 202.98080574532798436848679557442665100097656250;

Coefficients Interval 3

s1 = 1.28993207945813859183203931024763733148574829101562;
s2 = 0.67789608679455171103001021037925966084003448486328;
s3 = 1.07023243497406150837036875600460916757583618164062;
s4 = 1.7820832807415023513186724812840111553668975830078;
s5 = 3.42367689734088553876745208981446921825408935546875;
s6 = 6.9777525906397803723280048870947211980819702148437;
s7 = 14.9730443569138014225927690858952701091766357421875;
s8 = 31.472677428705704727462943992577493190765380859375;
s9 = 102.25556881992480384724331088364124298095703125;
s10 = -196.484231746760997339151799678802490234375;
s11 = 3893.28161386753390615922398865222930908203125;
s12 = -22060.1720108094523311592638492584228515625;
s13 = 1.03910745422446518205106258392333984375e5;
s14 = -2.684171832892236416228115558624267578125e5;
s15 = 3.703584434809651575051248073577880859375e5;

Coefficients Interval 4

s1 = 1.60046253674388494481206635100534185767173767089844;
s2 = 1.6004027497941937241421328508295118808746337890625;
s3 = 3.8839448184396876406765386491315439343452453613281;
s4 = 11.07596492047401426361830090172588825225830078125;
s5 = 35.48862996664502844623712007887661457061767578125;
s6 = 121.57268355404431758870487101376056671142578125;
s7 = 437.56835846512711896139080636203289031982421875;
s8 = 1557.36107423936709892586804926395416259765625;
s9 = 8094.959555248366086743772029876708984375;
s10 = -20311.7182554530081688426434993743896484375;
s11 = 8.073920855180737562477588653564453125e5;
s12 = -7.658388784700012765824794769287109375e6;
s13 = 6.206616511926199495792388916015625e7;
s14 = -2.72910558318305909633636474609375e8;
s15 = 6.4745996173633205890655517578125e8;

Coefficients Interval 5

S0 = -0.99999999999999988897769753748434595763683319091797;
s1 = -8.3333333333474549076136383973789634183049201965332e-2;
s2 = -1.8749999765584172972676668678104761056602001190186e-2;
s3 = -5.5803585831585957380274187755730963544920086860657e-3;
s4 = -1.89882855153310496805485474425267966580577194690704e-3;
s5 = -6.9982064277395383215113477959334886691067367792129e-4;
s6 = -2.6505393949424358931987733001278684241697192192078e-4;
s7 = -1.3635278961612355632000703753448078714427538216114e-4;