New Type of Non-Uniform Segmentation for Software Function Evaluation

Justine Bonnot, Daniel Menard, Erwan Nogues

New Type of Non-Uniform Segmentation for Software Function Evaluation

Justine Bonnot, Daniel Menard, Erwan Nogues
INSA Rennes
Rennes, France
Email: firstname.lastname@insa-rennes.fr

***Abstract*—Embedded applications integrate more and more sophisticated computations. These computations are generally a composition of elementary functions and can easily be approximated by polynomials. Indeed, polynomial approximation methods allow to find a trade-off between accuracy and computation time. Software implementation of polynomial approximation in fixed-point processors is considered. To obtain a moderate approximation error, segmentation of the interval I on which the function is computed, is necessary. This paper presents a method to compute the values of a function on I using non-uniform segmentation, and polynomial approximation. Non-uniform segmentation allows to minimize the number of segments created and is modeled by a tree-structure. The specifications of the segmentation set the balance between memory space requirement and computation time. The method is illustrated with the function $\sqrt{(}-\log(x))$ on the interval $[2^{-5};2^{0}]$ and showed a speed-up mean of 97.7 compared to the use of the library *libm* on the Digital Signal Processor C55x.**

I. Introduction

Technological progresses in microelectronics require the integration, in embedded applications, of numerous sophisticated processing composed of the computation of mathematical functions . To get the value of an intricate function, the exact value or an approximation can be computed. The challenge is to implement these functions with enough accuracy without sacrificing the performances of the application, namely memory usage, execution time and energy consumption. Several solutions can then be used. On the one hand, specific algorithms can be adapted to a particular function [5]. On the other hand, Look-Up Tables (LUT) or bi-/multi-partite tables methods can be used when low-precision is required. Nevertheless, the need to store the characteristics of the approximation in tables result in some impossibilities to include these methods in embedded systems because of the memory footprint. Finally, the majority of the proposed methods have been developed for a hardware implementation.

For the time being, two solutions exist for the software computation of these functions: the use of libraries such as *libm* that targets scientific computation and is very accurate but also quite slow, or the implementation of the CORDIC (COordinate Rotation DIgital Computer) algorithm that computes the approximation of trigonometric, hyperbolic and logarithmic functions.

In this work, the software implementation in embedded systems of mathematical functions is targeted as well as low cost and low power processors. To achieve cost and power constraints, no floating-point unit is considered available and processing is carried-out with fixed-point arithmetic. Then, the *polynomial approximation* of the function can give a very accurate result in a few cycles if the initial interval is segmented precisely enough. The proposed method is a new method of non-uniform subdivision. That new type of subdivision calls the Remez Algorithm of the Sollya tool [1] to approximate the function by polynomials on the segments obtained. The polynomial order is a trade-off between the computation error and the interval size. To obtain a given maximum approximation error, the polynomial order decreases, impliying the reduction of the segment size. This increases the number of polynomials to store in memory. For a given data-path word-length, the increase in polynomial order raises the fixed-point computation errors that annihilates the benefit of lower approximation error obtained by higher polynomial order. Thus, for fixed-point arithmetic, the polynomial order is relatively low. Consequently, to obtain a low maximal approximation error, the interval size is reduced. Accordingly, non-uniform interval subdivision is required to limit the number of polynomials to store in memory. The proposed method then originates two different types of error: the error of approximation $\epsilon_{app} = ||f - P||_{\infty}$ of the Remez algorithm (where f is the function to approximate and P is the approximating polynomial on a segment), and the error caused by the fixed-point computation ϵ_{fxp}.

The challenge of the proposed method is to find the accurate segmentation of the initial interval I. Different non-uniform segmentations [4] have been proposed. Nevertheless, they have only been implemented for hardware function evaluation and do not provide flexibility in terms of depth of segmentation.

The proposed method presents a segmentation of the interval I in a non-uniform way, using a tree-structure. The parameters of that method are the polynomial order N_d, the memory available for the application, the maximal computation time required and the precision needed on the approximation ϵ. The segmentation of I controls the error of approximation. The method provides the user Pareto curves giving the evolution of the memory size required depending on the computation time for a fixed precision, several degrees and several depths of segmentation. The user can then choose the most adapted degree and depth of segmentation given the constraints of the application. Besides, compared to table-based methods or the Cordic algorithm, our approach reduces respectively significantly the memory size required and the computation time.

The rest of the paper is organized as follows. First, the

related work is detailed in Section II. The method for obtaining the polynomial corresponding to an input value x is presented in Section III. The tool for approximating functions by polynomials using non-uniform segmentation is introduced in Section IV. The Binary Tree Determination is deepened in Section V. The Reduced Tree Determination is presented in Section VI. The experiments are shown in Section VII.

II. Related work

Several methods can be used to compute an approximate value of a function. Iterative methods as the CORDIC algorithm [6] are generally easy to implement. The CORDIC algorithm computes approximations of trigonometric, logarithmic or hyperbolic functions. That method offers the ability to compute the value using only shifts and additions and is particularly adapted to a processor with no floating-point unit. Nevertheless, that algorithm requires to store tables to compute the approximate value, and may require a lot of memory space as well as a long computation time. These latest depend on the precision required that sets the number of iterations as well as the size of the table to store. It is possible to compute the approximation of a function thanks to other methods whose algorithms are described in [5]. For instance, the Chebyshev orthogonal basis allows to approximate a function using only polynomials, if the approximation has to be done on $[-1; 1]$.

Hardware methods have been developed to compute approximate values of a function. The LUT method would consume the most memory space but would be the most efficient in computation time. That method consists in using 0-degree polynomials to approximate the value of the function. The initial segment is subdivided until the error between the polynomial and the real value is inferior to the maximal error on each segment obtained. However that segmentation has to be uniform, i.e. if the error criterion is not fulfilled on a single part of the initial segment, all the parts of the segments have to be segmented again. Improvements of the table-based method are the bi-/multi-partite methods detailed in [2]. The multi-partite method propose to segment the interval on which the function f has to be approximated to be able to approximate f by a sequence of linear functions. The initial values of each segment as well as the values of the offsets to add to these initial values to get whichever value in a segment have to be saved. The size of these tables has been reduced compared to bi-partite method exploiting symetry in each segment. That method offers quick computations and reduced tables to store but is limited to a low-precision required and to a hardware implementation.

Finally, another method has been developed for hardware function evaluation in [4]. The interval I is segmented to control the precision. On each segment, the function is approximated by the Remez algorithm. 4 different segmentation schemes are used: uniform segmentation or P2S (left, right or both) with segments increasing or decreasing by powers of two. Once a segmentation has been applied, if the error does not suit the criterion, the segment can be segmented again using one of these segmentation schemes. Afterwards, AND and OR gates are used to find the segment corresponding to an input value x. LUT are used to store the coefficients of the polynomials. Nevertheless, that method does not allow to control the depth of segmentation of I which is offered in the presented method and has been implemented for hardware function evaluation.

III. Computation of the approximating value

A. Method for Indexing the Polynomials

The initial interval I is beforehand segmented using a non-uniform segmentation that is stored in a tree structure T. The algorithm of segmentation is recursive: the Remez algorithm is called on a segment and the error of approximation $||f-P||_\infty$ is compared to the maximum error of approximation ϵ_{app}. If $||f-P||_\infty > \epsilon_{app}$, the segment is segmented and the algorithm is applied on each segment until the criterion is fulfilled on each segment. Each time the segmentation algorithm is called, a node is added in T to store the bounds of the created segment. The segmentation tree T is illustrated in figure 1.

Then, to each segment of the final segmentation corresponds an approximating polynomial. All the polynomials are stored in the table P whose lines correspond to the polynomials associated to the different segments. Consequently, the aim of this step is to determine for an input value x the index associated to the segment in which x is located. The problem is to find the path associated to the interval in a minimum time. The approach is based on the analysis and interpretation of specific bits of x formatted in fixed-point coding. The segment bounds are sums of powers of two, so these bits can be selected with a mask and interpreted as a binary number after having shifted them to align them on the LSB.

The tree storing the segmentation is not well balanced. For any tree level, the number of bits to analyze is not constant and depends on the considered node since all the nodes associated to a given level do not necessarily have the same number of children. Thus, the mask and the shift operations are specific for each node. Consequently, at each intermediate node n_{lj} (where l is the level and j the node) a mask and an offset between the index of the first child of a node and its index are associated. The mask is used to select the adequate bits of x to move from node n_{lj} to the next node $n_{l+1j'}$ located at level $l+1$.

The indexing method uses a bi-dimensional table $\mathcal{T}$ which stores for each node, the mask, the shift and the offset to pass from a level to the following. For the presented example in figure 1, the tree possesses 3 levels. Consequently, three tables are computed and presented in figure 2.

Each table contains as many lines as the total number of nodes in the level. This number is obtained by summing the number of nodes in the level to the number of leaves of the previous levels. Consequently, the number of information to store depends on the depth of the tree, the number of polynomials, and the degree of the approximating polynomials.

To find the shifts and masks values in the tables, for each level, the tree is observed. The value of the mask corresponding to a node is the binary conversion of the number of children of that node. The value of the shift corresponding is obtained substracting the number of children to the shift of the previous level.

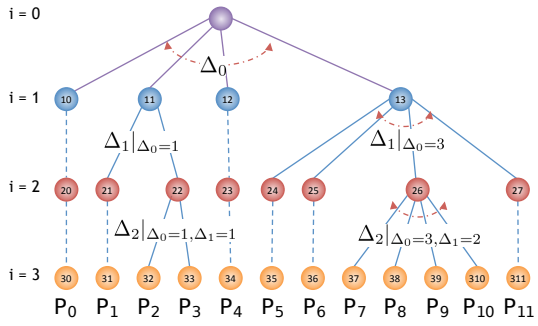
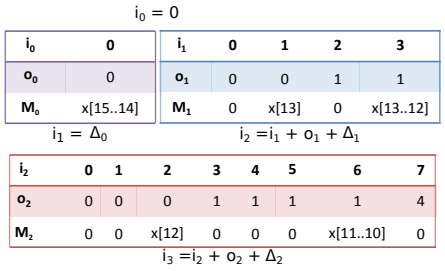


Fig. 1: Illustrating the segmentation method on a tree of depth 3



i_0	0
o_0	0
M_0	x[15..14]

i_1	0	1	2	3
o_1	0	0	1	1
M_1	0	x[13]	0	x[13..12]

i_2	0	1	2	3	4	5	6	7
o_2	0	0	0	1	1	1	1	4
M_2	0	0	x[12]	0	0	0	x[11..10]	0

Fig. 2: Indexing tables associated to the tree T in figure 1

B. *Fixed-point computation*

To improve the speed performances, the coefficients of the polynomials are formatted in fixed-point and stored in the bidimensional table $\mathcal{P}[i][d]$. The term i is the index of the interval and the coefficient is the one of the d-degree monomial. Each coefficient has its own fixed-point format to reduce the fixed-point computation errors. The computation of $P_i(x)$, where P_i is the approximating polynomial on the segment i, can be decomposed owing to the Horner rule, reducing the computation errors. According to the algorithm of Ruffini-Horner [7], each polynomial $P_i(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$ can be factored as:

$$P_i(x) = ((((a_n x + a_{n-1})x + a_{n-2})x + ...)x + a_1)x + a_0$$

Using that factorization, the calculation scheme can be decomposed in a basic loop kernel as in figure 3.

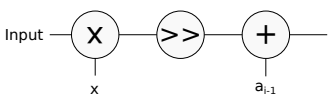


Fig. 3: The basic cell $n - i$ of the n-degree polynomial computation using the Horner rule

According to figure 3, shifts are necessary to compute the value of $P_i(x)$ in fixed-point coding. Indeed, the output of the multiplier can be on a greater format than necessary. By using interval arithmetic on each segment, the real number of bits necessary for the integer part can be adjusted with left shift operation.

Then, a right shift operation can be necessary to put the two adder operands on the same format. In our case, a quantification from double precision to single precision is done and creates a source of error. When the two shift values have been computed, they can be both added so as to do a single shift on the output of the multiplier that is stored in the bidimensional table $\mathcal{D}[i][d]$ where d corresponds to the iteration of the loop and i is the index of the segment.

IV. Tool for polynomial approximation

Numerous approximating methods have been developed with hardware function evaluation, and cannot consequently be integrated to a software function evaluation because of the memory space required. Consequently, a new method of software function evaluation is proposed in that paper. That method is composed of six different stages sharing data of the algorithm. To begin, the algorithm needs several input parameters:

1) The function to approximate f
2) The interval I on which f is approximated (f has to be continuous on I)
3) The maximum error of approximation ϵ, i.e. $\epsilon_{app}+\epsilon_{fp} < \epsilon$
4) The degree of the approximating polynomials N_d

These parameters are progressing through the diagram presented in figure 4 and the output of that diagram is the final tree giving the optimal segmentation of I. That final tree is determined after two stages: the binary tree T_{bin} has, in a first stage, to be determined, and then the user can specify a number of levels in the final tree. Then, another tree, $T_{reduced}$, with a reduced number of levels, is determined. In other words, the final tree is the one of the required depth. To get the final tree, several steps are then needed.

Firstly, the four inputs are needed in the first step of the approximation method, during the Binary Tree Determination. That step allows to segment I in a dichotomous way so that the function can be approximated, using the Remez Algorithm, on each segment obtained by that decomposition, while satisfying the error criterion. The Remez Algorithm is used in the approximation method to give the best approximating polynomials of the function on each segment, in the sense of the infinite norm and is called thanks to the Sollya tool [1]. The Binary Tree Determination is detailed in Section V. The obtained segmentation is saved in T_{bin} and sent to the Reduced Tree Determination where the degree of the approximating polynomials N_d is required too. A supplementary parameter is needed. Indeed, the determination of the binary tree gives the maximal depth of the segmentation, and the goal of that step is to reduce it: the number of levels required N_l has to be an input of that step. Reducing the depth of the segmentation allows to reduce the computation time as it is explained in the section VI. The output of that step is the Reduced Tree $T_{reduced}$.

Once that final tree is obtained, to compute the approximation of the value required, the segment in which the value to compute is has to be known. To determine the index of that segment the tables presented in Section III are used.

The error of the whole approximation, namely ϵ_{app} and ϵ_{fp} can then be computed. Finally the C code of the approximation is generated.

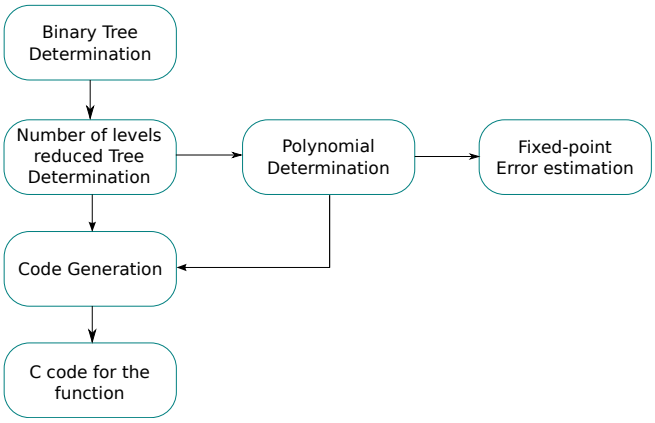


Fig. 4: The different stages of the algorithm for approximating function by polynomials

V. Binary Tree Determination

A. The Dichotomous Approach Segmentation

The dichotomous approach segmentation of the initial interval is presented in the algorithm 1.

Algorithm 1 The Polynomial Approximation

procedure RECURSE($Interval$)
  N_d; ϵ ; $I = [a; b]$;
  $P = Remez(f, I, N_d)$
  if $||f - P||_\infty \geq \epsilon$ **then**
    $I_1 = [a; \frac{b-a}{2} + a]$
    $I_2 = [\frac{b-a}{2} + a; b]$
    return RECURSE(I_1)
    return RECURSE(I_2)
  else
    return P
  end if
end procedure

The Sollya tool calls the Remez Algorithm on each segment while the error of approximation is higher than ϵ_{app}. Indeed, the user has specified the maximal error of approximation ϵ, so half of that error, ϵ_{app} is allocated to the Remez Algorithm and the other half to the error committed during the computation of the approximation in fixed-point coding ϵ_{fxp}. The algorithm is recursive: if the error criterion is not fulfilled on a segment, the segment is cut into two equal parts and the algorithm is applied on each obtained segment.

B. Computation of the Binary Tree

The segmentation of I is modeled by a tree T_{bin}. Each time the recursive algorithm is called, the bounds of the obtained segment are stored in a node of a tree structure. The interval bounds are sums of powers of 2 to ease the adressing. The root of that tree contains the bounds of I and the leaves the bounds of the segments on which the error criterion is verified, thus a polynomial is associated to each leaf.

The depth of the binary tree T_{bin} corresponds to the number of bits necessary to index the table $\mathcal{P}$. Then, in the binary tree, a bit is tested in each level of the tree. For instance, the binary tree obtained subdividing each segment in both equal parts is the deepest but leads to the minimal number of polynomials. On the contrary, the tree whose depth is 1 has the greatest number of polynomials. If the depth of the binary tree is N, then the number of polynomials in the tree of depth 1 is 2^N.

C. Binary Tree Determination on an example

The dichotomous approach segmentation is detailed in figure 5. The function $\sqrt{(} - log(x))$ is approximated on the interval $[2^{-5}; 1]$ with a specified error ϵ_{app} of 10^{-3} and 2-degree polynomials. According to the tree obtained, the function possesses a strong non-linearity in the neighbourhood of 1, since the tree branches particularly on this part of I. The correspondance between the polynomials P_i and the segments is located under the tree in figure 5.

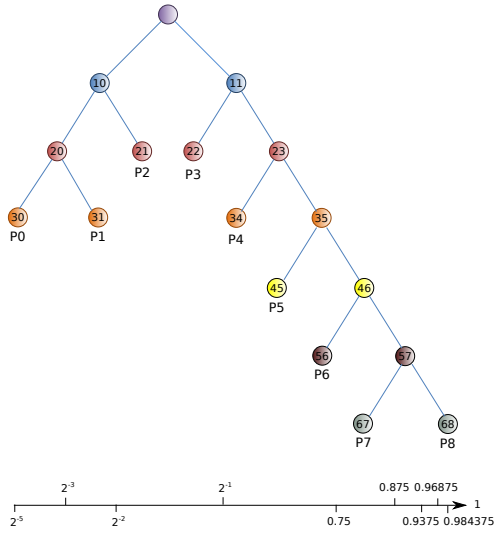


Fig. 5: T_{bin} corresponding to $f = \sqrt{(} - log(x))$, $\epsilon_{app} = 10^{-3}$, $I = [0.03125; 1]$ and $N_d = 2$

VI. Reduced Tree Determination

So as to minimize the computation time, the proposed method aims to optimize the dichotomous segmentation of I, reducing the number of levels in the tree modeling the segmentation. Indeed, the first step of the algorithm allows to know the number of bits to be tested to know in which segment the value x is. Then, the tree can be modified as long as the sum of the bits allocated to each level is equal to the depth of the binary tree T_{bin}. To know the best allocation of these bits, a tree of solutions T_{sol} is computed. It allows to minimize the time of computation as well as the hardware resources necessary to save tables $\mathcal{P}$ and $\mathcal{T}$.

A. The Tree of Solutions

Having the depth N_{lmax} of the binary tree T_{bin}, the number of bits to be tested to know in which interval the value x is, is known. The same number of bits has to be allocated to the reduced tree $T_{reduced}$. Besides, reducing the depth of the tree $T_{reduced}$, the memory space required to store the

polynomials (table $\mathcal{P}$) and their associated tables (table $\mathcal{T}$) does not have to increase. The tree of solutions studies all the possible allocation configurations and computes the theoretical required memory space depending on the allocation chosen. The depth of the tree of solutions is equal to the number of levels required and each path in the tree corresponds to an allocation configuration.

The tree computed in that step is the Tree of Solutions T_{sol}. Each node of that tree contains the number of bits allocated to the associated level of the node, and the leaves of the tree contain the value of the equation giving the required memory space to store tables $\mathcal{P}$ and $\mathcal{T}$. However, to compute T_{sol}, the Remez algorithm shall not be called to save time. Consequently, from T_{bin}, the list of the segments from the Dichotomous Segmentation is obtained and has to be compared to the one obtained with each allocation configuration. If a segment obtained with the new allocation is not included in one from the list of segments obtained from T_{bin}, then the error of approximation on that segment is bigger than the required error. Conversely, if a segment from the new segmentation is included in one from the list extracted from T_{bin}, then the error of approximation is smaller than the error required and that segment does not have to be segmented again.

Finally, to quickly know which path is the best given T_{sol}, the sole comparison of the values contained by the leaves is needed. A branch-and-bound algorithm can be used in that part to reduce the computation time for T_{sol} but is not needed since that computation is done upstream and not on the DSP. When the best allocation is found, the new tree $T_{reduced}$ is computed and the coefficients of the polynomials as well as the list of segments created are saved.

B. Reduced Tree Determination on an example

The Reduced Tree Determination is shown on the example given in Subsection V-C. The depth of the binary tree obtained in the Dichotomous Segmentation Step is 6. The tree of solutions obtained requiring $depth(T_{reduced}) = 3$ is in figure 6. The characteristics of the different allocation configurations at each node are given in the node itself: the number of bits allocated to the associated level are written in the node. The memory required by each allocation is indicated under each branch in bytes.

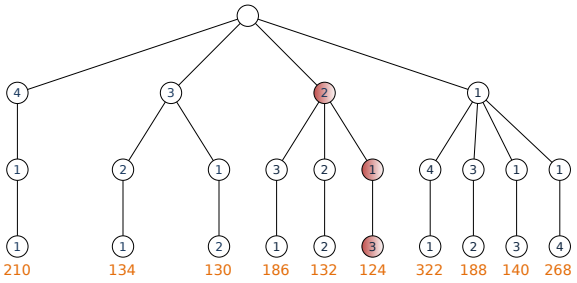


Fig. 6: T_{sol} with $f = \sqrt{(} - log(x))$ on $I = [2^{-5}; 1]$, with $\epsilon_{app} = 10^{-3}$ and $N_d = 2$

According to the memory space required by each path indicated in figure 6, the allocation configuration leading to the minimum memory space is the sixth path (nodes drawn in red in figure 6). To build $T_{reduced}$ of depth 3, 2 bits are allocated to the first level (the initial segment is subdivided in 4 segments), 1 bit is allocated to the second level, i.e. if needed (by comparing with the values obtained in the list of segments given by T_{bin}), the segments obtained in the previous level are subdivided in 2 parts. Finally, 3 bits are allocated to the last level, thus the same comparison is made with the list given by T_{bin}. If needed, the segments obtained in the previous level are subdivided in 8 parts.

On the example, $T_{reduced}$ obtained is represented in figure 7. The table detailing the segments corresponding to the polynomials is table I. Each bound of the segments can be written as a sum of powers of two, necessary to ease the addressing. As figure 7 shows it, 13 polynomials are created with that decomposition instead of 64 expected. That allocation leads then to the minimum memory space required to store both the polynomials coefficients and the tables to index them.

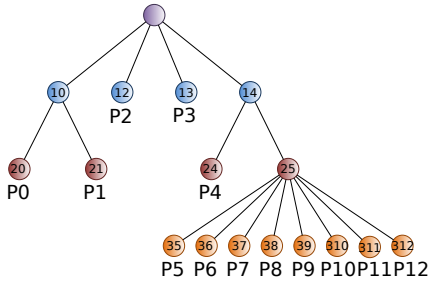


Fig. 7: $T_{reduced}$ with $f = \sqrt{(} - log(x))$ on $I = [2^{-5}; 1]$ with $\epsilon_{app} = 10^{-3}$ and $N_d = 2$

P_i	Associated segment
0	$[2^{-5}; 2^{-3}]$
1	$[2^{-3}; 2^{-2}]$
2	$[2^{-2}; 2^{-1}]$
3	$[2^{-1}; 0.75]$
4	$[0.75; 0.875]$
5	$[0.875; 0.890625]$
6	$[0.890625; 0.90625]$
7	$[0.90625; 0.921875]$
8	$[0.921875; 0.9375]$
9	$[0.9375; 0.953125]$
10	$[0.953125; 0.96875]$
11	$[0.96875; 0.984375]$
12	$[0.984375; 1]$

TABLE I: Correspondence between the polynomials P_i and the segments

VII. Experiments

The proposed method aims to minimize the computation time and the memory footprint to compute the approximated value of $f(x)$. Nevertheless, to know the optimal polynomial degree and the optimal depth of $T_{reduced}$ to suit the criteria of the processor targeted (in terms of memory space) and of the application (in terms of computation time), Pareto curves are needed. These curves provides the system designer with the optimal points for each polynomial degree tested, and take into account the number of levels in the tree as well as the required memory space that can be computed or the

computation time, knowing the size of the indexing table $\mathcal{T}$ and of the coefficients table $\mathcal{P}$ as well as the equation giving the computation time depending on the degree N_l and the number of levels N_l. These curves represent the memory space required depending on the depth of $T_{reduced}$ or the memory space required depending on the computation time in cycles. The functions chosen for the experiments are two composite functions (hardly computed with a basic CORDIC algorithm) and a trigonometric function to be able to compare the obtained results with an implementation of CORDIC.

Finally, the proposed method is compared to the use of *libm*, to an implementation of the CORDIC method and to the LUT method.

A. *Experiments with the function* $\sqrt{(}-\log(x))$

The curves are drawn in figures 8 and 9 for the function $\sqrt{(}-\log(x))$ approximated on $[2^{-5};2^{0}]$, allocating a maximal error for the Remez approximation of $\epsilon_{app} = 0.01$ and a maximal error for fixed-point coding of $\epsilon_{fxp} = 0.01$. The equation giving the computation time depending on the degree N_d and the number of levels N_l depends on the processor used and on the precision of fixed-point coding (single or double). These results have been obtained with a DSP C55x [3].

Experimentally, the only degrees that are adapted to the approximation of that function are 1 and 2. The maximal fixed-point coding error obtained with 1-degree polynomials, whatever the number of levels in the tree and in single precision, is $4.878 \cdot 10^{-4}$ and is consequently inferior to ϵ_{fxp}. Nevertheless, with 2-degree polynomials, in single precision, the maximal fixed-point coding error obtained is $6.21 \cdot 10^{-2} > \epsilon_{fxp}$. To be able to suit the fixed-point error criterion, data have to be coded in double precision. In double precision, with 2-degree polynomials, the maximal fixed-point coding error is $3.9 \cdot 10^{-3}$. 2-degree polynomials coded in double precision are adapted but higher degree polynomials lead to a too high fixed-point coding error.

Besides, the computation time has been determined given an equation depending on the fixed-point coding and on the processor, and determined experimentally. The equation giving the computation time depending on the degree N_d and the number of levels in the tree N_l with the provided C-code, in single precision on the target C55x, is:

$$t = 9 + 8 \cdot N_l + 3 \cdot N_d \tag{1}$$

In double precision, the coefficients to store raise dramatically, hence the preferable use of 1-degree polynomials in the case of the example:

$$t = 2 + 8 \cdot N_l + 63 \cdot N_d \tag{2}$$

Finally, the computation time using the library *libm* is constant and equal to 4528 cycles. The mean speed-up of the proposed method compared to the use of such a library is then with $N_d = 1$ equal to 98.82 and with $N_d = 2$ equal to 96.51.

Finally, the mean speed-up of the proposed method compared to the use of *libm* is 98.7 on the DSP C55x.

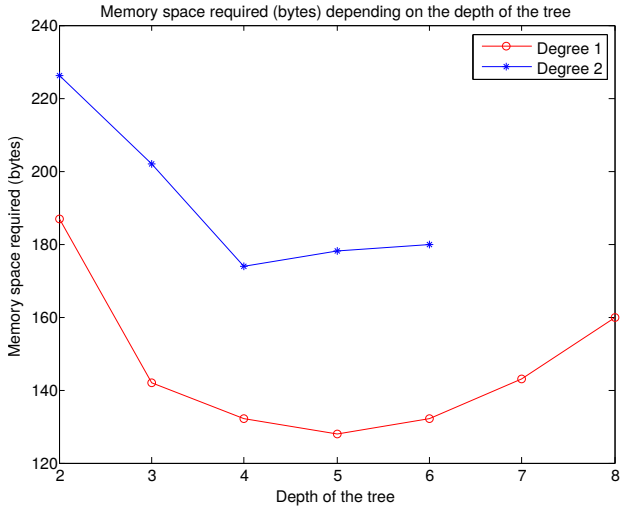


Fig. 8: Evolution of the memory space required to store the polynomials (table $\mathcal{P}$) and the shifts (table $\mathcal{D}$)(circles) and both polynomials ($\mathcal{P}$), shifts ($\mathcal{D}$) and the indexing table ($\mathcal{T}$) (diamonds) depending on N_l

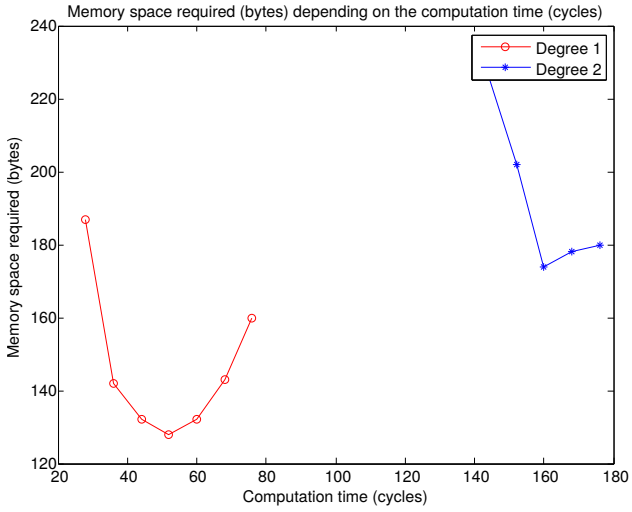


Fig. 9: Evolution of the memory space required to store the polynomials (table $\mathcal{P}$) and the shifts (table $\mathcal{D}$)(circles) and both polynomials ($\mathcal{P}$), shifts ($\mathcal{D}$) and the indexing table ($\mathcal{T}$) (diamonds) depending on t

B. *Experiments with the function* $\exp(-\sqrt{(}x))$

1) Experiments on DSP C55x: The function $\exp(-\sqrt{(}x))$ is studied on the interval $[2^{-6};2^{5}]$. The trees for 1-degree polynomials to 3-degree polynomials have been computed with depth varying from the maximal depth (binary tree) to a depth of 2. The evolution of the memory space required to store, on the one hand, polynomial coefficients table $\mathcal{P}$ and shifts table $\mathcal{D}$ for fixed-point computations S_{n-pol} and on the other hand both the polynomial coefficients table $\mathcal{P}$, the shifts table $\mathcal{D}$ and the indexing table $\mathcal{T}$ S_{n-tot}, depending on the number of levels in the tree N_l is drawn on figure 10. The trees are computed with a maximal error criterion ϵ_{app} of $5 \cdot 10^{-3}$. The

data and coefficients are on 16 bits, the fixed-point coding of the input is $Q_{6,10}$ and the tests have been computed on the DSP C55x .

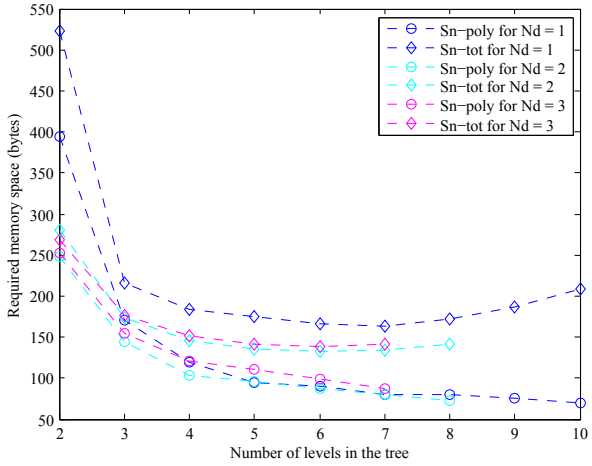


Fig. 10: Evolution of the memory space required to store S_{n-pol}(circles) and S_{n-tot}(diamonds)

The less levels the tree has, the greatest number of polynomials there is. Consequently, S_{n-poly} is high since the size of the tables $\mathcal{P}$ and $\mathcal{D}$ are high. Nevetheless, since the tree has a reduced depth, few indexing tables are needed (the depth of the table $\mathcal{T}$ is equal to the number of levels in the tree). On the contrary, the most levels the tree has, the less polynomials there are and S_{n-poly} is low since the size of tables $\mathcal{P}$ and $\mathcal{D}$ decreases. However, the table $\mathcal{T}$ is the biggest because of the number of levels in the tree.

Then, computing the total memory space required depending on the computation time for a required error provides the user with Pareto curves in figure 11 giving the optimal points for the computation time or the memory footprint. The function $\exp(-\sqrt(x))$ on the interval $[2^{-6}; 2^5]$ can be approximated by a polynomial of degree from 1 to 4 with datas coded on 16 bits, requiring a maximal total error (including ϵ_{app} and ϵ_{fxp}) of 10^{-2}. The maximum values of the error of fixed-point coding are in table II. A 5-degree polynomial does not suit this approximation since the error obtained with fixed-point coding is greater than the maximal error required.

Degree	ϵ_{fxp}
1	$[-2.8 \cdot 10^{-3}; 0]$
2	$[-2.5 \cdot 10^{-3}; 0]$
3	$[-2.5 \cdot 10^{-3}; 0.3 \cdot 10^{-3}]$
4	$[-2.4 \cdot 10^{-3}; 1.5 \cdot 10^{-3}]$
5	$[-2.7 \cdot 10^{-3}; 35.2 \cdot 10^{-3}]$

TABLE II: Intervals of the fixed-point coding error depending on the polynomial degree for the approximation of $exp(-\sqrt(x))$

According to the Pareto curves on figure 11, when the tree has a low number of levels, the required memory is high but the computation time is minimum. Then, the computation time increases with the number of levels of the tree while the required memory decreases until it reachs a minimum. After

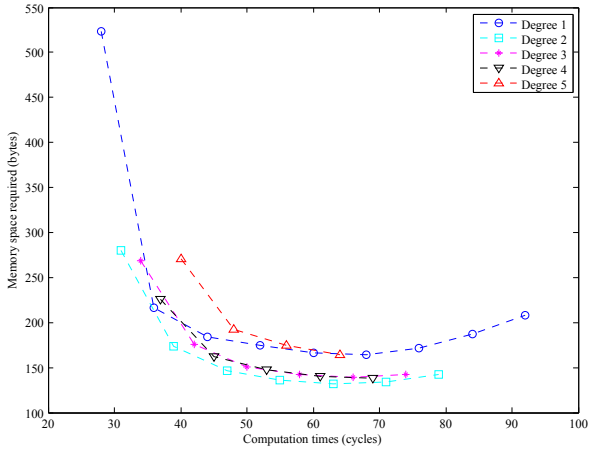


Fig. 11: The Pareto curves for the approximation of $exp(-\sqrt(x))$ on $[2^{-6}; 2^5]$)

that minimum for the required memory, it slightly tends to increase with the computation time and the number of levels.

2) Experiments on ARM Cortex M3: The same function is studied in the same approximation conditions on the target of ARM, the microcontroller Cortex M3. The Pareto curves representing the evolution of the memory space required depending on the computation time are in figure 12.

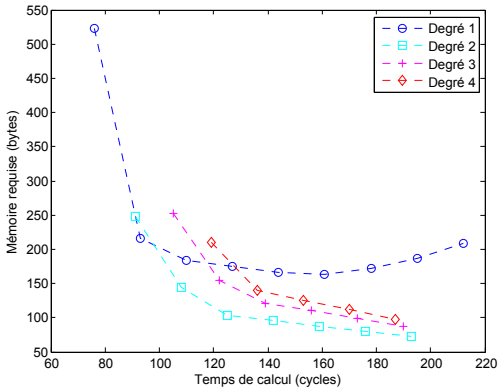


Fig. 12: The Pareto curves of the memory space required depending on the computation time for the approximation of $\exp(-\sqrt(x))$ on Cortex M3

The computation time of the approximation of the value $f(x)$ using the proposed method and data formatted in single precision on the Cortex M3 can be obtained thanks to the following equation determined experimentally:

$$t = 27 + 17 \cdot N_l + 15 \cdot N_d \quad (3)$$

C. Experiments with the function $\sin(x)$

The proposed method is tested with a trigonometric function to compare the computation time and the memory space required to the one obtained using the CORDIC method. The

value to compute is $\sin(\frac{\pi}{3})$. The segment on which the function is approximated is $[0; \frac{\pi}{2}]$. The depth of T_{bin} is then, with 1-degree polynomials, 3. The total error of approximation is $\epsilon = 0.01$. With 2-degree polynomials the binary tree has a depth of 1. The results obtained are presented in the table III.

Degree	Depth	Memory(bytes)	Time(cycles)
1	2	38	28
1	3	42	36
2	1	16	23

TABLE III: Results obtained approximating $sin(x)$ on $[0; \frac{\pi}{2}]$ with an error of 0.01 and the proposed method

So as to compare the CORDIC method to the proposed method on a trigonometric function ($\sin(x)$), the curve of the computation time depending on the precision required (figure 13) and the curve of the memory space required depending on the precision required (figure 14) are drawn. Two degrees of polynomial approximation are tested.

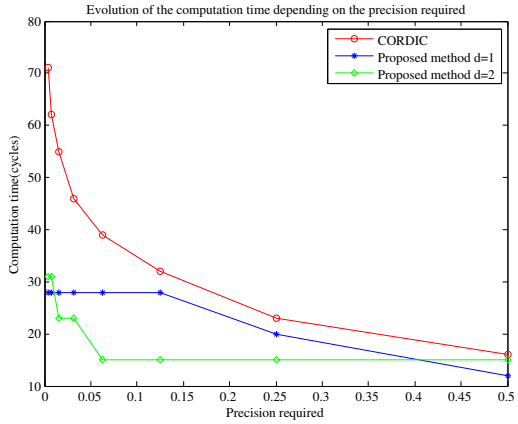


Fig. 13: The evolution of the computation time depending on the precision required for the CORDIC method, the proposed method with 1-degree and 2-degree polynomials

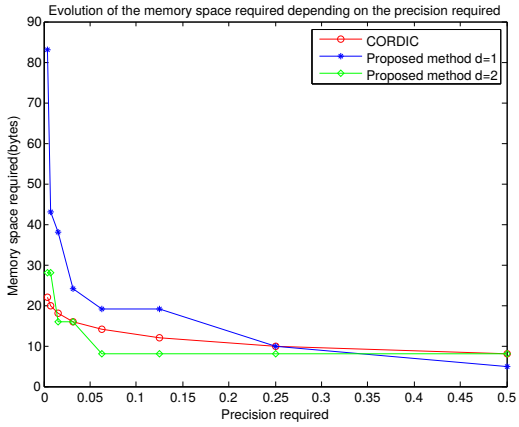


Fig. 14: The evolution of the memory space required depending on the precision required for the CORDIC method, the proposed method with 1-degree and 2-degree polynomials

The results obtained show that the computation time is always lower for the proposed method. Besides, until a precision of 0.01, the proposed method consume less memory space (with 2-degree polynomials) than the CORDIC method.

D. Comparison of the memory space required by the proposed method and by the table-based method

Function	ϵ_{tot}	I	Mem_{tab}	Mem_{prop} (mean)
$\exp(-\sqrt{(}x))$	10^{-2}	$[2^{-6}; 2^{5}]$	8192	530
$\sqrt{(}-\log(x))$	0.02	$[2^{-5}; 1]$	256	225
$\sin(x)$	10^{-2}	$[0; \frac{\pi}{2}]$	256	84

TABLE IV: Comparison of the memory required for approximating the functions below using the proposed method Mem_{prop} and using the table-based method Mem_{tab}

VIII. Conclusion

The non-uniform segmentation scheme followed by polynomial approximation proposed in that paper provides the system designer with Pareto curves giving the optimal points for the memory footprint or the computation time. With these Pareto curves, the system designer can then choose the degree N_d and the depth of the tree saving the non-uniform segmentation that suit the best the targeted application. Besides, the proposed method has been compared to other existing methods: in terms of software implementation, the methods used currently are the CORDIC method or libraries such as *libm*. Compared to the CORDIC method, the proposed method is always faster and if the parameters of the approximation are well chosen, consumes less memory. The proposed method has also a mean speed-up equal to 97.7 on the DSP C55x compared to the use of *libm*. Finally, compared to a hardware method such as the LUT method, the gain in memory is significant.

References

[1] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

[2] F. de Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, March 2005.

[3] Texas Instruments. *C55x v3.x CPU Reference Guide*. Dallas, June 2009.

[4] D.-U. Lee, R.C.C. Cheung, W. Luk, and J.D. Villasenor. Hierarchical segmentation for hardware function evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):103–116, January 2009.

[5] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press New York, NY, USA, 1992.

[6] Jack E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, September 1959.

[7] Edmund Taylor Whittaker and George Robinson. *The calculus of observations*. Blackie London, 1924.