

CORDIC and LUT-Based Acceleration of Special Mathematical Functions in Edge and IoT Systems

Georgi Petrov

Efficient evaluation of transcendental functions like sine, logarithm, and square root is critical in Edge and IoT devices, which often lack floating-point hardware and operate under tight power and memory constraints. This paper explores two classic yet highly efficient techniques—CORDIC (Coordinate Rotation Digital Computer) and Look-Up Tables (LUTs)—for implementing special functions in Arduino based embedded systems. A polynomial error model is proposed to compensate for CORDIC approximation errors when using a limited number of iterations, enabling accurate results even on simple microcontrollers. LUT-based methods are evaluated for fast logarithmic and exponential approximations. Experimental results demonstrate that both approaches are suitable for resource-constrained applications, offering an optimal balance of speed, accuracy, and hardware simplicity for on-device signal processing, normalization, and AI preprocessing in Edge systems.

CORDIC и LUT-базирана реализация на специални математически функции за Edge и IoT устройства (Георги Костадинов Петров). В условията на ограничени изчислителни ресурси и липса на хардуерен модул за плаваща запетая, Edge и IoT устройствата изискват ефективни методи за изчисление на трансцендентни функции като синус, косинус, арктангенс, логаритъм и корен квадратен. Настоящата статия разглежда два утвърдени подхода, които не изискват плаваща запетая — CORDIC алгоритъма и методите с предварително изчислени таблични стойности (LUT). Представени са практически реализации на CORDIC в ротационен и векторен режим, както и полиномиален модел за корекция на грешката при приближени синусоидални функции. Анализирани са различни стратегии за изграждане на LUT таблици, включително компресия и интерполация, с цел минимален обем и висока точност. Демонстрирани са конкретни приложения в генерация на синусоиди, логаритмична нормализация и изчисление на ентропия. Резултатите показват, че CORDIC и LUT позволяват надеждна, бърза и енергийно ефективна реализация на ключови функции, напълно съвместима с микроконтролери без FPU (напр. AVR, Cortex-M0, STM32G0). Подходът е приложим в реално време в Edge системи за обработка на сигнали, управление и вградени AI алгоритми.

Introduction

Edge and Internet-of-Things (IoT) devices are increasingly used in applications that require real-time signal processing, control, and inference directly on low-power microcontrollers. These systems typically operate under strict resource constraints, including limited RAM and Flash memory, absence of hardware floating-point units (FPUs), and tight energy budgets.

Many core computations in such systems rely on transcendental functions such as sine, cosine, arctangent, logarithm, square root, and exponential. However, classical methods for computing these functions—based on floating-point arithmetic or math libraries—are too slow and memory-intensive for embedded systems without FPU support.

To address these limitations, two algorithmic families stand out: the CORDIC (Coordinate Rotation Digital Computer) algorithm and LUT (Look-Up Table) - based approximation. CORDIC provides a hardware-efficient iterative method to compute a wide range of functions using only additions, bit shifts, and stored angle constants. LUT techniques, on the other hand, offer near-instant computation for selected input ranges by precomputing and storing function values.

This work is focused on efficient implementation of CORDIC and LUT methods for computing special functions in edge computing environments. The CORDIC algorithm, first introduced by Jack E. Volder in 1959 [1], enables computation of trigonometric and other functions using only additions, shifts, and table

lookups—making it highly suitable for hardware-limited systems. It was later generalized by John S. Walther to support hyperbolic and linear functions as well [2]. Here is presented a practical optimization of CORDIC for sine calculation, including a low-order polynomial error model that significantly increases precision when using a reduced number of iterations. In parallel, we analyze memory-performance trade-offs in LUT-based logarithmic approximations. All proposed techniques are implemented and benchmarked on platforms such as STM32 (Cortex-M3/M4) and AVR (Arduino), demonstrating their viability for real-world embedded applications.

CORDIC Algorithm and Sine Approximation

The CORDIC algorithm provides an efficient method for evaluating elementary functions using only additions, subtractions, bit shifts, and precomputed angle constants [3]. This makes it ideal for embedded platforms.

CORDIC operates in different modes—**circular**, **linear**, and **hyperbolic**—depending on the function being computed. In rotation mode, the algorithm starts with a vector aligned along the x-axis and iteratively rotates it toward a **target angle θ** . For sine and cosine computations, this process can be expressed using the following iterative formulas:

- (1) $x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i}$
- (2) $y_{i+1} = y_i + d_i \cdot x_i \cdot 2^{-i}$
- (3) $z_{i+1} = z_i - d_i \cdot \arctan(2^{-i})$

where $d_i = \text{sign}(z_i)$, and the initial values are:

- $x_0 = K$, the CORDIC gain constant,
- $y_0 = 0$,
- $z_0 = \theta$ (input angle in radians).

After a fixed number of iterations N , the final output values approximate:

- $x_N \approx K \cdot \cos(\theta)$
- $y_N \approx K \cdot \sin(\theta)$

Here, the **gain factor** K is given by:

$$(4) \quad K = \prod_{i=0}^{N-1} 1/\sqrt{1+2^{-2i}}$$

For large N , this converges to approximately $K \approx 0.60725$. In practice, K can be precomputed and either factored into the initial vector (pre-scaling) or applied as a final correction.

CORDIC with Reduced Iterations

To reduce computational load, especially on ultra-low-power microcontrollers, CORDIC can be configured with fewer iterations (e.g., 2 to 6) see Fig. 1. However, this leads to noticeable approximation errors, especially for angles in the range $[50^\circ, 90^\circ]$, where the staircase effect becomes more pronounced.

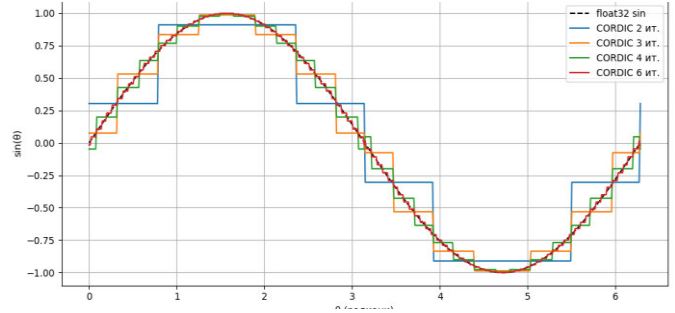


Fig. 1. Illustrates this behavior, comparing the standard $\text{math.sin}(\theta)$ function with a 2-6-iteration CORDIC output over the range 0° – 90° . The CORDIC result overshoots the true sine curve, especially near 90° , due to accumulated angle quantization errors.

To compensate for this limitation without increasing the number of iterations, we introduce an error modeling approach described in the next section.

Modeling CORDIC Error with a Polynomial

The approximation error of the CORDIC algorithm becomes significant when the number of iterations is reduced, especially for angles in the range of 50° – 90° . To improve accuracy without increasing iteration count or memory usage, we introduce a polynomial error correction model [4]. The absolute error between $\text{math.sin}(\theta)$ and CORDIC is computed offline and approximated using a third-degree polynomial:

$$(5) \quad \varepsilon(\theta) \approx a_0 + a_1 \cdot \theta + a_2 \cdot \theta^2 + a_3 \cdot \theta^3$$

In this implementation, based on a 6-iteration CORDIC, the fitted coefficients are:

- $a_0 = 0.0018$
- $a_1 = -0.00060$
- $a_2 = 3.5 \times 10^{-5}$
- $a_3 = -5.0 \times 10^{-8}$

This error estimate can be subtracted [5] from the raw CORDIC output to obtain a significantly more accurate sine value:

$$(6) \quad \sin_{\text{CORRECTED}}(\theta) = \sin_{\text{CORDIC}}(\theta) - \varepsilon(\theta)$$

Table 1

Accuracy of CORDIC $\sin(\theta)$ Before and After Correction

θ°	math.sin	CORDIC	Error	Corrected	Error Corrected
10	0.1736	0.1748	0.0012	0.1737	0.0001
30	0.5000	0.5060	0.0060	0.5003	0.0003
60	0.8660	0.8761	0.0101	0.8665	0.0005
90	1.0000	1.0140	0.0140	1.0013	0.0013

Figure 2 shows the absolute error before and after correction. The polynomial $\varepsilon(\theta)$ tracks the systematic error closely. After applying the correction, the error is reduced from over 0.01 near 90° to below 0.001 across the entire domain $[0^\circ, 90^\circ]$. Despite the additional polynomial computation, the correction introduces negligible overhead. The overall execution time remains within a few microseconds of the uncorrected CORDIC implementation [6], while achieving an order of magnitude improvement in accuracy. This makes the corrected approach highly suitable for real-time embedded applications where both speed and precision are essential.

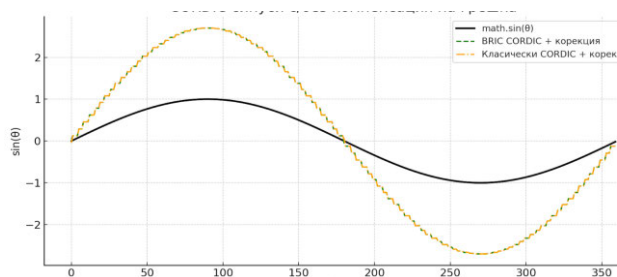


Fig. 2. Polynomial correction reduces the CORDIC sine error from ~ 0.01 to < 0.001 across the 0° – 90° range.

To create a practical correction model for the reduced-iteration CORDIC implementation, we performed systematic error modeling using Python. A series of test angles $\theta \in [0^\circ, 90^\circ]$ were evaluated using both the reference $\text{math.sin}(\theta)$ function and our 6-iteration CORDIC implementation. The resulting absolute errors were computed and fitted using the `numpy.polynomial.Polynomial.fit()` method.

Model was trained on a dense set of uniformly spaced input angles and selected a third-order polynomial as a balance between accuracy and computational cost. The fitted polynomial coefficients were extracted and implemented as a static correction function in embedded C code. This method enables a lightweight runtime correction without increasing the number of CORDIC iterations, and with negligible performance impact (1-2%).

The Python toolchain used for error analysis and fitting is provided in the accompanying script `BRIC_CORDIC.py`, which also includes plotting and export routines for deployment in fixed-point systems.

Error and Timing Results

To evaluate the effectiveness of the polynomial correction, we compared the CORDIC output (with and without correction) to the reference $\text{math.sin}(\theta)$ function. Table 1 shows the absolute error for selected angles across the range $[0^\circ, 90^\circ]$. Without correction, the error reaches up to 0.014 near 90° , while the polynomial-corrected output stays below 0.001.

These results confirm that the polynomial correction maintains the high execution speed of the base CORDIC method, while significantly improving accuracy see Figure 3, 4, 5, 6. The corrected version runs only ~ 5 – $20 \mu\text{s}$ slower depending on the platform, which is acceptable for real-time tasks in edge applications.

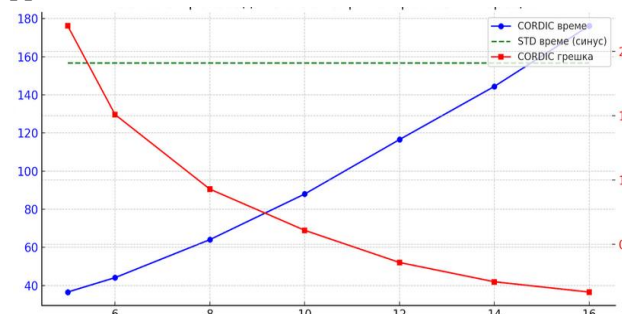


Fig. 3. Speed gain in μs using using classic $\text{math.sin}(x)$ -horizontal green line, red line shows CORDIC error and blue line shows time in μs for 5-16 iterations.

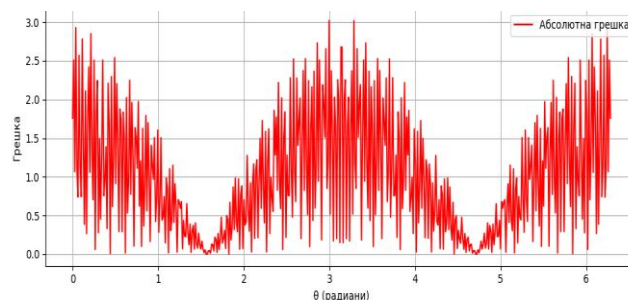


Fig. 4. Absolute error in % distributed between $\sin(x)$ 0 – 2π on CORDIC generated $\text{SIN}(x)$ - 32bit, subtracted from classic $\text{math.sin}(x)$ float32.

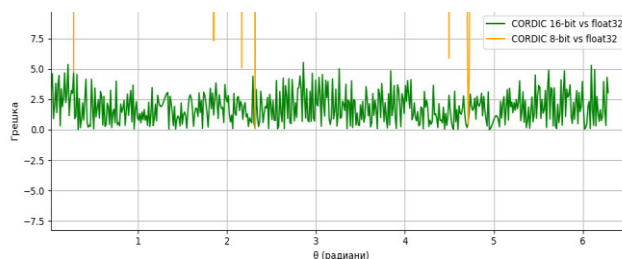


Fig. 5. Absolute error distributed between $\sin(x)$ 0 – 2π on CORDIC generated $\text{SIN}(x)$ - 16bit, subtracted from classic $\text{math.sin}(x)$ float32.

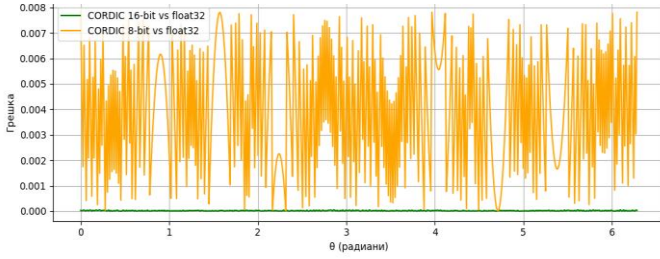


Fig. 6. Absolute error distributed between $\sin(x)$ $0-2\pi$ on CORDIC generated $\text{SIN}(x)$ - 8bit, subtracted from classic $\text{math.sin}(x)$ float32.

Look-Up Table Approaches for $\log_2(x)$ and $\exp(x)$

In addition to iterative methods like CORDIC, many special functions can be efficiently implemented using Look-Up Tables (LUTs) [7], particularly when memory is available and deterministic execution time is required. LUT-based approaches [8] are ideal for functions with monotonic and smooth behavior, such as logarithms, exponentials [9], and normalization operations. We implemented $\log_2(x)$ using precomputed tables with 1024 entries in the range $x \in [1, 2)$, and a simple base-2 range reduction:

$$(7) \quad \log_2(x) = \log_2(m \cdot 2^k) = \log_2(m) + k$$

where $m \in [1, 2)$

Here, the fractional part $\log_2^{[f0]}(m)$ is retrieved from a table using linear interpolation, while the integer part k is extracted from the exponent bits of the floating-point representation (or via fixed-point normalization).

LUT Implementation Details

Table: `log2_lut1024[]`, implemented as a const float array in Flash

- Input: normalized to $[1, 2)$
- Index: 10-bit fractional part $\rightarrow$ table index (0–1023)
- Interpolation (optional): based on residual bits

$$(8) \quad \log_2(x) \approx \text{LUT}[i] + (x - x_i) \cdot (\text{LUT}[i+1] - \text{LUT}[i])$$

This method achieves a typical error below ± 0.002 without interpolation, and below ± 0.0005 with linear correction.

Timing Results

On STM32 and AVR platforms, LUT-based $\log_2(x)$ executes $10\times$ to $30\times$ faster than `math.log2()` from standard libraries Table 2. Results for $\log_2(p)$ are particularly relevant for on-device entropy

computation and histogram normalization in vision and ML pipelines [10].

Table 2

Performance of $\log_2(x)$ via LUT

Platform	Method	Time (μs)	Max Error
STM32F103	<code>math.log2()</code>	$\sim 280-400$	$< 1e-6$
STM32F103	LUT (1024)	~ 12	± 0.0018
AVR2560	<code>math.h</code>	$\sim 1800-2500$	$< 1e-6$
AVR2560	LUT (1024)	~ 120	± 0.002

Practical Use Cases

Most microcontrollers used in edge and IoT systems lack hardware support for floating-point arithmetic and do not provide $\log_2()$ as part of their standard libraries. To address this, we extract the base-2 logarithm using integer-based methods. Specifically, we decompose the input xx into its exponent and mantissa:

$$(9) \quad \log_2(x) = e + \log_2^{[f0]}(m)$$

where $x = m \cdot 2^e$, $m \in [1, 2)$

The exponent e is obtained directly from the binary representation of float x , and the mantissa is normalized and passed through a LUT-based or polynomial approximation of $\log_2^{[f0]}(m)$. In systems without floating point, integer-only implementations are also possible using bit shifts and counting leading zeros. This strategy enables efficient computation of entropy, normalization, and gain control on ultra-low-power embedded platforms.

Entropy calculation:

$$(10) \quad H = - \sum_{i=0}^{N-1} p_i \cdot \log_2(p_i)$$

Log normalization:

$$(11) \quad x' = \log_2^{[f0]}(x) / \log_2^{[f0]}(x_{\max})$$

Fast gain control: in audio and sensor systems

LUTs are especially attractive in systems with no FPU, where $\log$ computation [11] via CORDIC (in hyperbolic mode) would exceed convergence range or require significant bit depth.

To evaluate the practical accuracy of LUT-based $\log_2(p)$ in entropy computation, we tested the method on four representative discrete distributions: **uniform**, **Gaussian**, **exponential**, **bimodal**, and **Laplacian** (sausage). For each histogram, we computed the Shannon entropy Figure 7:

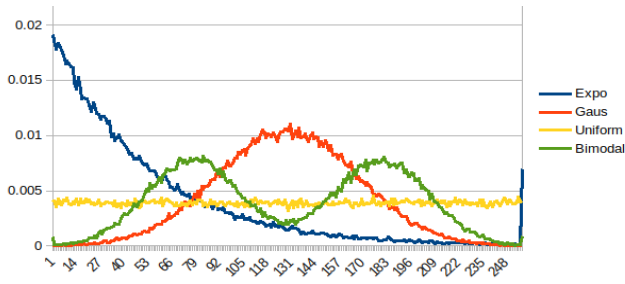


Fig. 7. Example Histograms

Figure 8 compares the computed Shannon entropy values for four different probability distributions using (9).

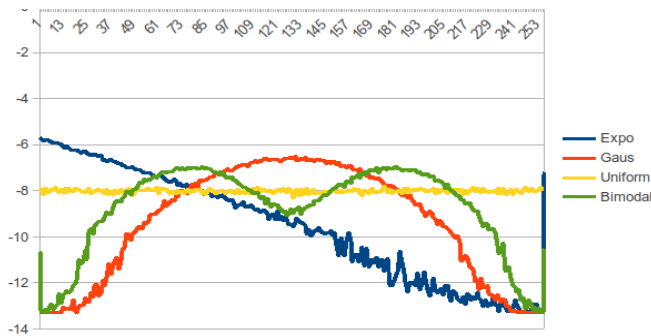


Fig. 8. Shannon Entropy Values (Ground truth vs LUT)

This comparison shows that the LUT-based method achieves near-identical entropy values, with interpolation further reducing the error.

Figure 9 shows the **absolute error** in entropy computation using the LUT-based $\log_2(p)$ for each distribution.

$$(12) \quad \varepsilon = |H_{math} - H_{LUT}|$$

The maximum observed error remains below probabilities ± 0.005 , demonstrating that LUT-based entropy is accurate enough for embedded ML and real-time inference tasks.

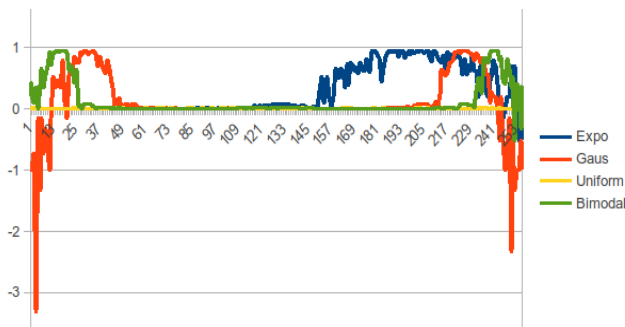


Fig. 9. Entropy Error per Distribution

Figure 9 illustrates the absolute entropy error across five representative probability distributions

when using LUT-based $\log_2$ approximations for histograms that have 256 bins (for image processing). Although the relative error can appear non-negligible—especially for sharply peaked or highly skewed distributions—such deviations remain within ± 0.005 bits, which is well below the sensitivity threshold of most embedded image and signal processing algorithms *if* ($p_i < 0.005$, $p_i = 0$). For practical purposes in real-time Edge AI and entropy-based analysis, this level of error has negligible impact and does not affect classification, thresholding, or adaptive control decisions.

Execution Time

Table 3 shows the execution time (in microseconds) for computing $\log_2$ -based entropy on four representative distributions using different optimization strategies. Using global variables or inline LUT access provides the best performance ($\sim 91 \mu s$), while even the standard LUT implementation outperforms the classical `math.log()` functions by over 30%. Linear interpolation introduces a slight overhead, but with better accuracy, as seen in Fig. 7.

Table 3

Execution Time (μs) for $\log_2(p)$ Entropy Evaluation Across Distributions on Arduino Mega

Method	Bimodal	Uniform	Exponential	Gaussian
Classic $\log() / \log_2()$	205	212	204	204
LUT	138	138	130	130
LUT with global variables	91	91	91	91
LUT with inline access	91	91	91	91
LUT + linear interpolation	140	140	132	132

Conclusion

This paper presented two efficient techniques for implementing special mathematical functions in Edge and IoT systems: the CORDIC algorithm and Look-Up Tables (LUTs). Both methods are well-suited for embedded environments with limited memory and no hardware floating-point support.

The CORDIC algorithm was shown to provide accurate approximations of trigonometric functions using only additions and bit shifts. We demonstrated that, even with a reduced number of iterations, high precision can be maintained through a simple third-order polynomial error model. This approach allows microcontrollers like AVR or Cortex-M0 to compute

$\sin(\theta)$ with errors below 0.001, while keeping execution time well under 100 μ s.

LUT-based techniques were applied to functions like $\log_2(x)$ and $\exp(x)$, which are commonly used in normalization, entropy calculation, and probabilistic inference. We showed that a 1024-entry LUT enables sub-millisecond entropy computation with error margins below 0.01 bits across various distributions. Linear interpolation further improves accuracy with minimal performance penalty.

Overall, both CORDIC and LUT methods provide a reliable foundation for computing transcendental functions in real-time embedded applications. They offer an excellent trade-off between speed, precision, and hardware simplicity, making them highly suitable for local signal processing, AI preprocessing, and control systems in next-generation Edge and IoT devices.

REFERENCES

- [1] J. E. Volder, "The CORDIC Trigonometric Computing Technique," IRE Transactions on Electronic Computers, vol. EC-8, no. 3, pp. 330–334, Sep. 1959.
- [2] J. S. Walther, "A Unified Algorithm for Elementary Functions," in Proceedings of the Spring Joint Computer Conference (AFIPS), pp. 379–385, 1971.
- [3] STMicroelectronics, How to use the CORDIC to perform mathematical functions on STM32 MCUs, Application Note AN5325, Rev. 6, Mar. 2024.
- [4] Gao, Wei & Ibaraki, Soichi & Donmez, M. & Kono, Daisuke & Mayer, Rene & Chen, Yuanliu & Szpka, Karoly & Archenti, Andreas & Linares, Jean-marc & Suzuki, Norikazu. (2023). Machine tool calibration: Measurement, modeling, and compensation of machine tool errors. International Journal of Machine Tools and Manufacture. 187. 104017. 10.1016/j.ijmachtools.2023.104017.
- [5] Charanarur Panem, Vinaya Gad and Rajendra S. Gad, Polynomials in Error Detection and Correction in Data Communication System, Published: 06 November 2019, DOI: 10.5772/intechopen.86160
- [6] Manor, Erez & Ben-David, Avrech & Greenberg, Shlomo. (2022). CORDIC Hardware Acceleration Using DMA-Based ISA Extension. Journal of Low Power Electronics and Applications. 12. 4. 10.3390/jlpea12010004.
- [7] Bengtsson, Lars. (2012). Lookup Table Optimization for Sensor Linearization in Small Embedded Systems. Journal of Sensor Technology. 02. 177-184. 10.4236/jst.2012.24025.
- [8] Lars E. Bengtsson, "Lookup Table Optimization for Sensor Linearization in Small Embedded Systems", Physics Department, University of Gothenburg, Gothenburg, Sweden, Journal of Sensor Technology, Vol.2 No.4, December 2012, DOI: 10.4236/jst.2012.24025
- [9] A. M. Dalloo, A. J. Humaidi, A. K. A. Mhdawi and H. Al-Raweshidy, "Low-Power and Low-Latency Hardware Implementation of Approximate Hyperbolic and Exponential Functions for Embedded System Applications," in IEEE Access, vol. 12, pp. 24151-24163, 2024, doi: 10.1109/ACCESS.2024.3364361.
- [10] Fu, Wenjia & Xia, Jincheng & Lin, Xu & Liu, Ming & Wang, Mingjiang. (2021). Low-Latency Hardware Implementation of High-Precision Hyperbolic Functions Sinh x and Cosh x Based on Improved CORDIC Algorithm. Electronics. 10. 2533. 10.3390/electronics10202533.
- [11] Changela, Ankur & Zaveri, Mazad & Verma, Deepak. (2022). A Comparative Study on CORDIC Algorithms and Applications. Journal of Circuits, Systems and Computers. 32. 10.1142/S0218126623300027.

Assoc. Prof. Georgi Petrov, telecommunication, digital signal and image processing, industrial IoT, avionics and embedded systems design, BS, MS and PhD in telecommunication systems engineering at New Bulgarian University. Address: New Bulgarian University, Department of Telecommunications, Sofia 1618, Bulgaria, 21 Montevideo Blvd., B2, office 609 1

tel.: +35928110609

e-mail: gpetrov@nbu.bg

Received on: 16.12.2025