Embedded genetic algorithm for low-power, low-cost, and low-size-memory devices

Denis R. da S. Medeiros1 | Matheus F. Torquato2 | Marcelo A. C. Fernandes1,3,4

1Laboratory of Machine Learning and Intelligent Instrumentation, nPITI-IMD, Federal University of Rio Grande do Norte, Natal, Brazil
2College of Engineering, Swansea University, Swansea, UK
3Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte (UFRN), Natal, Brazil
4John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, Massachusetts, USA

Correspondence
Marcelo A. C. Fernandes, Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte, Natal, Rio Grande do Norte, Brazil. Email: mfernandes@dca.ufrn.br

Funding information
Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Grant/Award Number: Finance Code 001

Peer Review
The peer review history for this article is available at https://publons.com/publon/10.1002/eng2.12231.

Summary
This work proposes a strategy to create an embedded genetic algorithms (GAs) for low-power, low-cost, and low-size-memory devices. This strategy aims to provide the means of GAs to run as a low-cost and low-power consumption embedded system, where microcontrollers (μCs) are commonly used. The implementation details are presented, emphasizing the limitations and restrictions imposed to turn it more compact and efficient. In addition, data related to the algorithm effectiveness, processing time, and memory consumption were obtained from simulations, oscilloscope measurements, and using the hardware-in-loop technique. Finally, this implementation is compared with other implementation from the literature and the results show that 8-bits μCs can run GAs for several practical applications.

KEYWORDS
embedded system, genetic algorithms, low-cost, low-power, low-size-memory, microcontrollers

1 | INTRODUCTION

Artificial intelligence (AI) is a field that will offer many opportunities for emerging markets and services and will revolutionize almost every segment of society. AI covers a wide range of activities: Medical, multimedia, finance, or advertising, among others. The results are applied to the most different systems, such as clusters, mobile phones, or even smart sensors. However, AI requires intensive processing and high-performance hardware, and this can be a problem on IoT applications in which the target is hardware with low-power, low-cost, and low-size-memory.
On the other hand, low-power, low-cost, and low-size-memory devices, such as microcontrollers (μCs), have been used in applications. Microcontrollers have been used in several types of applications in areas such as industrial automation, control, measuring, consumer electronics, and so on. In addition, there is a growing demand for the use of these devices, mainly in emerging markets such as Internet of things (IoT), smart grid, and machine to machine, for instance. Hardware platforms such as μCs can be classified as digital systems called systems on chips (SoCs), which are commonly used as embedded systems for specific applications. Usually, this type of SoC is composed by a 8, 16, or 32-bits general purpose microprocessor, program, and data memory, besides being coupled with other peripherals, such as counters, signal generators, analog-to-digital converters, among others. Even though μCs have a limited processing power, their main advantages are the low-power consumption, reduced size, and price, which makes them appropriate to be used in several situations where these characteristics are required, such as in IoT applications, for example, References 1-4.

The use of intelligent systems in embedded hardware with μCs have been studied and there are a few works in the literature on that subject. This can be explained by the fact that AI is currently one of the most studied areas in computing and most applications make use of it to solve different types of problems. Nevertheless, some AI techniques demand high computational power, especially when applications impose tight time constraint. Therefore, it has been a challenge to optimize and adapt AI algorithms to make them suitable to run on hardware platforms such as the μCs. As a result, several research groups have been working in this direction. 5-8

Among the AI techniques, the genetic algorithms (GAs) are a kind of bioinspired metaheuristics used to solve search and optimization problems in several areas of industry and engineering. The GAs are based on the Darwin’s theory of evolution and they provide efficiency in solving numerous different kinds of problems. However, they have a high computational complexity, which is affected by the population size and the evaluation function (also called fitness function).

There are some works in the literature that use GAs on μCs, such as References 11-14. Nonetheless, the GA implementations presented on most of these works try to solve specific problems, hence their results do not focus on the GA performance or on its implementation itself. For example, References 13 and 14 show the processing time only for a small number of individuals (or chromosomes) in their GA implementation. On the other hand, in References 11 and 12 the GA implementation is not optimized for low-power processors, as those that are commonly found on μCs. In fact most of these projects do not provide details about the implementation and do not analyze the processing time and memory consumption used by the GA for different settings.

The GA algorithm encodes the information on a string of bits, and this codification permits to use a bitwise operation in most algorithms steps. The use of bitwise operations is used in dedicate hardware proposals but it is have not been used to low-power, low-cost, and low-size-memory devices. Bitwise operations have been used in several works in the literature to reduce the algorithm processing time. 16-19

Therefore, the main goal of this present work is to present a GA implementation for low-power, low-cost, and low-size-memory devices, such as μCs, with the purpose of enabling its use in a generic way for different kinds of applications. The bitwise operations are used in all GA steps, and the proposal was optimized to 8-bits μCs architecture. This optimization allows the integration with other 16-bits and 32-bits embedded systems architectures.

An embedded GA implementation with these characteristics is suitable for applications where there is a need to solve some nonlinear optimization or search problem but with limitations in terms of costs and mainly power consumption. For example, a battery-powered drone or robot that needs to calculate the shortest path to reach a destination avoiding obstacles using GAs. Another example is the use of GAs embedded in automobiles to calibrate car engine parameters as presented in References 23,24. In Reference 24, the authors were using even an 8-bit microcontroller to run the experiments. Finally, for these types of applications, the number of individuals and the number of generations do not need to be large—in Reference 21, for instance, the number of individuals and the number of generations were at most 30. Therefore, a low-cost and low-power μC is feasible for these cases.

The implementation details, results concerning the processing time and memory consumption are analyzed for different parameters, such as population size and the number of bits needed to represent each individual from the population, the number of generations, and the type of evaluation function (or fitness function). Based on these results, it is possible to identify limits for each parameter in order to keep the GA feasible, as well as the parts of algorithm which affects more the performance and consume more resources. Furthermore, the hardware-in-loop (HIL) technique is used in order to
validate the correct operation of the GA. Finally, a comparison with other implementation available in the literature is also made to show how this one performs.

2 | GENETIC ALGORITHMS

GAs are iterative algorithms which start by randomly generating a population of \( N \) individuals (or chromosomes) that are mapped into \( M \) bits each. For each \( k \)th iteration of the algorithm, called generation, the \( N \) individuals (or chromosomes) pass through the operations of evaluation, selection, crossover, and mutation. After these operations, a new population with the same size is generated and its individuals can replace all or some individuals from the previous generation and then this new created population becomes the starting point of the next generation. This cycle is repeated \( K \) times, where \( K \) is a GA parameter and represents the number of generations in which the algorithm will be executed.

The Algorithm 1 represents the pseudocode of the GA proposed in this work and it is inspired in Reference 10. The vector \( x_j(k) \) represents the \( j \)th individual of the \( N \)-sized population \( X(k) \), on the \( k \)th generation. Each \( j \)th individual has dimension \( D \), thus the element \( x_{ji}[M](k) \) represents the \( i \)th dimension of this individual, which is mapped into \( M \) bits. Therefore, the population \( X(k) \) can be expressed as

\[
X(k) = \begin{bmatrix} x_0(k) & \cdots & x_{N-1}(k) \end{bmatrix} = \begin{bmatrix} x_{0,0}[M](k) & \cdots & x_{0,D-1}[M](k) \\ \vdots & \ddots & \vdots \\ x_{N-1,0}[M](k) & \cdots & x_{N-1,D-1}[M](k) \end{bmatrix}.
\]

(1)

The first step of the algorithm is the generation of the first population (Line 1 of Algorithm 1). After this, the evaluation function (or fitness function), named EF (Line 4 of Algorithm 1), is applied overall \( N \) individuals \( x_j[M](k) \) and it calculates the fitness value for each one. The index of the best individual is stored in \( j_B \) to be used in the elitism operation later. The fitness of the \( j \)th individual with dimension \( D \) is stored in \( y_j[B](k) \), where \( B \) is the number of bits required to represent the fitness. The better the fitness value \( y_j[B](k) \) of the individual \( x_j[M](k) \), the greater the probability of this individual to be selected or forwarded to the next generation. These values of the \( N \) individuals are stored as follows

\[
y(k) = \begin{bmatrix} y_0[B](k) \cdots & y_{N-1}[B](k) \end{bmatrix}.
\]

(2)

After the evaluation, the next operation is the selection, which aims to choose the individuals with best fitness value \( y_j[B](k) \) in order to generate an improved population compared with the previous one. There are several selection methods described in the literature such as the roulette wheel selection, the stochastic universal sampling, the tournament selection, and the rank-based selection, for example, Reference 25. In this implementation, the tournament selection is used since it is one of the most used and efficient methods according to Reference 26. This method compares two or more individuals randomly selected from \( X(k) \). In other words, this method compare the fitness value \( y_j[B](k) \) of those chosen individuals and the best one, called the tournament winner, moves on to share its genes in the next step, that is, the crossover operation.

The selection function is represented in the pseudocode as SF (Line 10 of Algorithm 1). Its inputs are the vector \( y(k) \) and the matrix \( X(k) \) of the \( k \)th generation and its output is the index of the \( j \)th individual, \( x_k(k) \), that was selected. The elitism technique can also be applied, so that the best \( E \) individuals of the current population are passed directly to the new population. In this implementation, \( E = 1 \) and the best individual is placed on the first position of the new population (Line 16 of Algorithm 1).

The next operation is the crossover, where two or more individuals from the current population, \( X(k) \), are combined to generate new ones that will be inserted into the new population, \( X(k+1) \), after passing through the mutation operation. In the literature, there are several strategies for the crossover such as the one-point crossover, two-point, and uniform.27 The implementation developed in this work allows the use of any of these three options by configuring a mask, as will be shown later in the Section 3.5.6.

The crossover function is defined as CF (Line 10 of Algorithm 1) and has as input two indices, which represents two individuals, and the matrix \( X(k) \) of the \( k \)th generation, which represents the current population. As output, it returns two new individuals generated by the crossover, which will be part of the offspring of the \( k \)th generation. This offspring is
stored into the matrix \( Z(k) \), which is defined as

\[
Z(k) = \begin{bmatrix}
z_0(k) & \cdots & z_{N-1}(k)
\end{bmatrix} = \\
\begin{bmatrix}
z_{0,0}[M](k) & \cdots & z_{0,D-1}[M](k) \\
\vdots & \ddots & \vdots \\
z_{N-1,0}[M](k) & \cdots & z_{N-1,D-1}[M](k)
\end{bmatrix}.
\] (3)

With the matrix \( Z(k) \) filled with \( N \) individuals, the following operation is the mutation, where \( P \) individuals will have their information randomly modified. In this work, the mutation function is defined as \( MF \) (Line 13 of Algorithm 1) and receives as input a new individual \( z_v[M](k) \), from the \( k \)th generation, and replaces it with its modified version. The mutation rate, called \( RM \), defines the proportion of individuals that suffer mutation, hence \( P \) can be specified as

\[
P = \lfloor RM \times N \rfloor.
\] (4)

---

**Algorithm 1.** Genetic Algorithm Pseudocode

▷ Generation of the initial population
1: Initialize(\( X(0) \))
    ▷ Starts to process the generations
2: for \( k \leftarrow 0 \) to \( K - 1 \) do
    ▷ Calculates the fitnesses and evaluates the individuals (or chromosomes)
3:     for \( j \leftarrow 0 \) to \( N - 1 \) do
4:         \( y_j[B](k) \leftarrow EF(x_j(k)) \)
5:         if \( y_j[B](k) < y_{jb}[B](k) \) then
6:             \( jb = j \)
7:         end if
8:     end for
    ▷ Selection and crossover
9:     for \( i \leftarrow 0 \) to \( N - 1 \) with step 2 do
10:        \[
        \begin{bmatrix}
z_i(k) \\
z_{i+1}(k)
        \end{bmatrix} \leftarrow CF \left( \begin{bmatrix} SF(y(k),X(k)) \\ SF(y(k),X(k)) \end{bmatrix} \right)
        \]
11:     end for
    ▷ Mutation
12:     for \( v \leftarrow 0 \) to \( P - 1 \) do
13:         \( z_v(k) \leftarrow MF(z_v(k)) \)
14:     end for
    ▷ Elitism
15:     for \( i \leftarrow 0 \) to \( D - 1 \) do
16:         \( x_{0,i}[M](k) \leftarrow x_{jb,i}[M](k) \)
17:     end for
    ▷ Updates the population
18:     for \( j \leftarrow 1 \) to \( N - 1 \) do
19:         for \( i \leftarrow 0 \) to \( D - 1 \) do
20:             \( x_{j,i}[M](k) \leftarrow z_{j,i}[M](k) \)
21:         end for
22:     end for
23: end for

The last operation of the GA is the population update. In the literature, there are different approaches in which the entire older population or only a part of it is replaced.\(^{28}\) In this implementation, the entire population \( X(k) \) is renewed, that is, each \( j \)th individual of the \( k \)th generation is replaced by a new individual, generating the population of the next generation, \( X(k+1) \). These new individuals can come from both the offspring of the \( k \)th generation, stored in \( Z(k) \), or directly from the old population due the elitism technique (Lines 16 and 20 of Algorithm 1).
3 | IMPLEMENTATION

The implementation proposed in this work aims to produce an efficient and optimized design to run on 8-bits microcontrollers. Thus, after its compilation, it should have small size, low memory occupation, efficient use of variables in order to save data memory, and optimized operations in an effort to decrease the processing time of the GA. Therefore, to meet these requirements it was needed to define some development strategies and constraints that are going be to explained in the following sections.

3.1 | Parameters constraints

Based on the pseudocode presented on Algorithm 1 and in order to achieve the goal of saving memory and reducing processing time, several parameters constrains were defined. Moreover, it is important to mention that this work was implemented using the C programming language, but the same ideas can be applied to other programming languages as well.

Initially, as shown in the previous section, there are three main variables in the GA are \(X(k)\), \(y(k)\) and \(Z(k)\) and they are responsible for consuming most of data memory. Thus, the first established constraint was related to the representation of the individual, where the number of bits, \(M\), needs be the same of the primitive data types \(\text{uint8_t}\), \(\text{uint16_t}\), and \(\text{uint32_t}\). In other words, the individuals have to be represented as unsigned integer numbers of 8, 16, or 32 bits, respectively. In order to standardize the source code, it was created an alias responsible to represent the individuals data type. It is called \(\text{chromosome}_t\), which is configured before the compilation and it helps the code to be more organized and legible.

The second constraint was to limit the population size \(N\) and the number of generations \(K\). \(N\) has to be a power of 2, less or equal to 256 and \(K\) needs to be less or equal to 256. By using these limits, it was possible to optimize some bitwise operations during the selection operation, which will be detailed in Section 3.5.5, as well as to reduce the size of auxiliary variables that are used during the loops. The types of these auxiliary variables were standardized by the aliases \(\text{popsize}_t\), which indexes the size of the population, and \(\text{generationsize}_t\), which indexes the generations. Both aliases are mapped into \(\text{uint8_t}\), since 8-bits are sufficient to represent 256 values. Mathematically, these limits are defined as

\[
N = 2^{A_1}, \text{ where } 1 \leq A_1 \leq 8, \quad (5)
\]

and

\[
1 \leq K \leq 256. \quad (6)
\]

Other four aliases were also created: \(\text{chromosomesize}_t\), \(\text{dimensionsize}_t\), \(\text{fitness}_t\), and \(\text{normalization}_t\). Variables of the type \(\text{chromosomesize}_t\) saves the size of the individual in bits, \(M\), which can be 8, 16, or 32 bits long as explained before, and it is represented as \(\text{uint8}_t\). The second alias, \(\text{dimensionsize}_t\), saves the number of dimensions that an individual can have and it is represented as \(\text{uint8}_t\) as well. The last two aliases are related to the evaluation function, which is likely to deal with real numbers, and therefore can be represented as floating point numbers, such as IEEE-754 single or double precision, for example. Thus, \(\text{fitness}_t\) is represented with \(B\) bits and \(\text{normalization}_t\) with \(G\) bits, which can vary depending on the compiler implementation.

From the constraints presented above, it can be calculated the number of bytes that the main variables of the program will need from the data memory during the execution. Hence, the amount of consumed memory by the matrix \(X(k)\), the vector \(y(k)\) and the matrix \(Z(k)\) is defined for each one, respectively, as

\[
n_{\text{RAM}}^X = \frac{M \times N \times D}{8} \text{ bytes}, \quad (7)
\]

\[
n_{\text{RAM}}^y = \frac{B \times N \times D}{8} \text{ bytes}, \quad (8)
\]

and

\[
n_{\text{RAM}}^Z = \frac{M \times N \times D}{8} \text{ bytes}. \quad (9)
\]
Finally, based on Equations (7) to (9), it is possible to estimate the total data memory consumed by the entire GA. Given that \( n_{\text{aux}}^{\text{RAM}} \) is the amount of memory in bytes consumed by auxiliary variables, such as those used inside of the scope of loops, for instance, this amount can be expressed as

\[
\begin{align*}
\text{n}_{\text{RAM}}^{\text{AC}} &= \text{n}_{\text{RAM}}^{\text{X}} + \text{n}_{\text{RAM}}^{\text{Y}} + \text{n}_{\text{RAM}}^{\text{Z}} + \text{n}_{\text{RAM}}^{\text{aux}} = \\
&= \frac{2(M \times N \times D) + (B \times N \times D)}{8} + \text{n}_{\text{aux}}^{\text{RAM}}.
\end{align*}
\]

(10)

3.2 GA parameters representation

One important characteristic of the GAs is their parameters, such as number of individuals, maximum number of generations, mutation rate, and so on, which must be chosen as per the problem where the GA is being applied. Since all these parameters are fixed values, the idea in this implementation is to avoid wasting memory with variables to store them, thus the idea is to insert them directly in the processor instructions.

An automatic and straightforward way to perform this insertion is to use the preprocessing directives from the C programming language. Essentially, these directives are expressions evaluated during the precompilation stage and occurs, as its name suggests, before the compilation of the program itself. A common directive is the macro, which is a type of key that is replaced by some code fragment in the source code and that is created using the reserved word define. Other useful directives are the conditionals, as with them it is possible to evaluate expressions and define parts of the source code that can be enabled or disabled, for example. Among them can be cited the if, elif and the else, for example.

Based on these concepts, the GA parameters were defined as macros in the source code. Due to the existence of constraints that require some numbers to be less than 256, depending on the value, it can be represented as an immediate in the assembly instruction during the compilation step. Moreover, the implementation was configured in such a way that some operations are processed differently according to those parameters defined previously. An example of how this strategy works is shown in Algorithm 2.

Algorithm 2. Example of the use of directives

- Examples of macros

1. (DIRECTIVE) #define M 16
2. (DIRECTIVE) #define N 32
3. (DIRECTIVE) #define D 2
4. (DIRECTIVE) #define K 32
5. (DIRECTIVE) #define P 2
- Examples of conditionals

6. (DIRECTIVE) IF M == 32
7. typedef uint32_t chromosome_t;
8. (DIRECTIVE) #define MASK 0xFFFF0000
9. (DIRECTIVE) ELSE IF M == 16
10. typedef uint16_t chromosome_t;
11. (DIRECTIVE) #define MASK 0xFFF0
12. (DIRECTIVE) ELSE
13. typedef uint8_t chromosome_t;
14. (DIRECTIVE) #define MASK 0xF0
15. (DIRECTIVE) END IF
- Definition of the matrix \( X(k) \)
16. chromosome_t X[N][D];
- Loop for processing the generations of the GA
17. for k ← 0 to K − 1 do
18. [· · ·]
19. end for
From now on, in the following algorithms it must be considered that the GA parameters were already defined as macros. They are the following ones:

- \( M \) — Size of the individual (number of bits);
- \( D \) — Number of dimensions of the individuals;
- \( N \) — Population size (number of individuals);
- \( K \) — Maximum number of generations;
- \( P \) — Number of individuals that will mutate;
- \( L_{\text{min}} \) — Minimum value used in the normalization;
- \( L_{\text{max}} \) — Maximum value used in the normalization.

### 3.3 Pseudorandom number generator

During the first experiments using this implementation, it was noticed that disregarding the evaluation function, which varies according to the problem, the parts of the algorithm that were more time consuming were those ones that needed to use a function to generate a pseudorandom number, a pseudorandom number generator (PRNG). In the first implementation of this work, the PRNG was implemented using the function \( \text{rand} \), present in the \texttt{stdlib C} standard library, and which generates a pseudorandom integer number.\(^{29}\)

The function \( \text{rand} \) is a general-purpose function that can be used for most applications but is not optimized to run on \( \mu \)Cs. In most implementations of the C compiler, this function works internally with 16 or 32-bits numbers and it consumes excessive clock cycles on a 8-bits processor, for example. Thus, in order improve this pseudorandom number generation, other alternatives described in the literature were considered and tested for this implementation.\(^{31-33}\)

The goal of testing different PRNG was to find one that could generate a long enough sequence of pseudorandom numbers consuming the less clock cycles as possible on an 8-bits \( \mu \)C. After testing several of these algorithms, the most efficient in the results was the linear feedback shift register generator (LFSRG), which consumed about 50\% less clock cycles than the \( \text{rand} \) function.

In order to make the LFSRG more suitable for this work, since the individuals can assume sizes of 8, 16, or 32 bits, it was implemented three optimized versions of this generator. The function \( \text{lfsr\_rand32}() \) is used in the case \texttt{chromosome\_t} needs 32 bits during the population initialization, for example. In the rest of the program it is only used the \( \text{lfsr\_rand8} \), since the constraints ensure the population size is not bigger than 256. Finally, each function was defined with the modifier \textit{inline} so that in exchange of consuming more program memory, where it is possible to save some clock cycles by avoiding the function calling.

- \texttt{inline uint32\_t lfsr\_rand32(\text{void})}, returns a 32-bits pseudorandom number;
- \texttt{inline uint16\_t lfsr\_rand16(\text{void})}, returns a 16-bits pseudorandom number;
- \texttt{inline uint8\_t lfsr\_rand8(\text{void})}, returns a 8-bits pseudorandom number.

Furthermore, it was also implemented three functions responsible for receiving the initial number of the PRNG, usually refereed as the PRNG seed in the literature. This seed can come from an external device such as a sensor, for example, to make sure there is real entropy added to the generator. These functions are shown below and each one must be used together with its respective function previously listed.

- \texttt{void\_t lfsr\_srand32(uint32\_t seed)}, receives a 32-bits seed;
- \texttt{void\_t lfsr\_srand16(uint16\_t seed)}, receives a 16-bits seed;
- \texttt{void\_t lfsr\_srand8(uint8\_t seed)}, receives a 8-bits seed.

### 3.4 Logical shift

Another part of the implementation that was identified spending a significant amount of clock cycles was the logical shift operation that is used in the mutation operation, as will be explained in Section 3.5.8. During the first tests, the logical
shift with 16 or 32-bits numbers was requiring a high number of clock cycles. By definition, the instructions and buses present in an 8-bits μCs are able to deal with 8-bits data. That means that operations with 16 or 32-bits variables takes more than one clock cycle to be completed because they need to be divided into several steps.

These details are relevant because, as explained in the parameters constrains section, the individuals in this GA implementation can be represented in 8, 16, or 32 bits. With that in mind, the assembly code generated after the compilation was analyzed and it was noticed that the logical shift works recursively. This means that when the program needs to shift positions of a 32-bits number, for example, it needs to shift one bit per time successively until it reaches the correct position.

Thus, in order to avoid logical shift operations with 16 or 32-bits numbers, a different strategy was adopted based on the characteristics of the bit shift that appears in the mutation operation, which consists in moving the number 1 to a specific position. That consists in identifying the byte or octet where the final position of the number 1 will appear and then, to do a simple 8-bits logical shift inside of that byte. For example, by admitting the number 1 needs to be shifted 27 bits to the left, then it would be located in the 4th byte. This is equivalent to shift this number 3 positions inside the 4th byte, for example, as shown in Figure 1.

The Algorithm 3 shows how this strategy was implemented. As it can be seen, conditionals directives are used in such way that this strategy comes in only when the GA is working with 16 or 32-bits individuals, that is, when $M = 16$ or $M = 32$. Even though this idea uses more instructions and consumes more program memory, it was able to reduce the clock cycles in about 65% in the worst case, that is, when it needs to shift the number 1 to the 31th position.

### Algorithm 3. Binary Shift Algorithm

```plaintext
1: (DIRECTIVE) IF M == 32
2: if (position < 8) then
3:     result ← (0x00000001 ≪ position)
4: else if (position < 16) then
5:     result ← (0x00000100 ≪ (position − 8))
6: else if (position < 24) then
7:     result ← (0x00010000 ≪ (position − 16))
8: else
9:     result ← (0x01000000 ≪ (position − 24))
10: end if
11: (DIRECTIVE) ELSE IF M == 16
12: if (position < 8) then
13:     result ← (0x0001 ≪ position)
14: else
15:     result ← (0x0100 ≪ (position − 8))
16: end if
17: (DIRECTIVE) ELSE
18: result ← (0x01 ≪ position)
19: (DIRECTIVE) END IF
```

### 3.5 Modularization

In order to make the implementation proposed in this work easy to read and modify, the different steps of the GA were divided into several modules. In addition, other auxiliary modules were defined as well and each one of them is named as follows:

![Figure 1](image-url)
Figure 2  Illustration of the GA architecture and its modules.

GA, genetic algorithm

- Initialization function module (IFM);
- Normalization function module (NFM);
- Evaluation function module (EFM);
- Fitness function module (FFM);
- Selection function module (SFM);
- Crossover function module (CFM);
- Selection and crossover function module (SCFM);
- Mutation function module (MFM);
- Update function module (UFM);
- New population function module (NPFM).

The architecture that describes the whole GA and how those modules are connected is shown in Figure 2. In this figure, it can be seen the chronological order wherein each module comes in and where the variables \(X(k), y(k),\) and \(Z(k)\) are used. It is important pointing out that only the reference to these variables are passed through the modules to avoid memory waste. In the following subsections the implementation of each module will be detailed as well as their processing time will be informed.

3.5.1  Initialization function module

The first module of the program represents the initialization of the population. Its only input is the reference for the matrix \(X(0)\) and its values are initialized with random numbers which follow a uniform distribution. The function \(\text{RNG-IFM()}\) represents a PRNG which produces a \(M\)-bits number. The structure of this module is show in Algorithm 4.
Algorithm 4. IFM Algorithm
1: function IFM(X(0))
2:    popsize_t j
3:    dimensionsize_t i
4:    for j ← 0 to N − 1 do
5:        for i ← 0 to D − 1 do
6:            xj[M](0) ← RNG-IFM()
7:    end for
8: end function

\(c_{\text{clk}}^{\text{attr}}\) is the number of clock cycles needed by the processor to make an \(M\)-bits attribution, \(c_{\text{clk}}^{\text{for}}\) to make an iteration of the for loop, and \(c_{\text{clk}}^{\text{RNG–IFM}}\) for the function generate a \(M\)-bits pseudorandom number. Therefore, the total processing time of this module measured in seconds, \(t_{\text{IFM}}\), can be defined as

\[
t_{\text{IFM}} = N \times \left[ c_{\text{clk}}^{\text{for}} + D \times (c_{\text{clk}}^{\text{for}} + c_{\text{clk}}^{\text{attr}} + c_{\text{clk}}^{\text{RNG–IFM}}) \right] \times \frac{1}{\text{CLK}},
\]

where \(N\) is the population size, \(D\) the number of dimensions of an individual and \(\text{CLK}\) is the frequency of operation of the processor measured in Hz.

3.5.2 Normalization function module

This second module is where the values of the individuals, \(x_j[M]\), are normalized to a real number between the boundaries, \(L_{\text{min}}\) and \(L_{\text{max}}\), in order to limit the search region. These limits are defined before compiling the program and the normalization can be described as

\[
\bar{x}_i(k) = L_{\text{min}} + \left( \frac{L_{\text{max}} - L_{\text{min}}}{2^M - 1} \right) \times x_i(k),
\]

where \(\bar{x}_i(k)\) represents the normalized individual, from the type \text{normalization_t}, and \(x_i(k)\) the original individual, which is defined as a \(M\)-bits integer number.

In the Algorithm 5, it can be seen how the normalization module works, so that it is applied to all individual dimensions. Furthermore, it is possible to calculate the processing time for this module as

\[
t_{\text{NFM}} = D \times \left( c_{\text{clk}}^{\text{for}} + c_{\text{clk}}^{\text{attrG}} + c_{\text{clk}}^{\text{CalcNorm}} \right) \times \frac{1}{\text{CLK}},
\]

which is proportional to the number of dimensions, \(D\), the number of clock cycles for a loop iteration, \(c_{\text{clk}}^{\text{for}}\), an attribution of \(G\) bits, and the calculus of the normalization, represented by \text{CalcNorm}. Again, \(\text{CLK}\) represents the processor frequency.

Algorithm 5. NFM Algorithm
1: function NFM(xj(k), \(x(k)\))
2:    dimensionsize_t i
3:    for i ← 0 to D − 1 do
4:        \(\bar{x}_i(k) \leftarrow L_{\text{min}} + (L_{\text{max}} - L_{\text{min}}) \times x_j(k)/(2^M - 1)\)
5:    end for
6: end function
3.5.3 | Evaluation function module

This module represents the evaluation function of the GA, which is the main link between this metaheuristic and the
problem which the GA is being applied to. It consists of a function \( f(\cdot) \), the evaluation function of an optimization
problem, and has as input a normalized individual, \( \vec{x}(k) \), and produces as output a value of type \textit{fitness}_t. This value represents
the fitness degree which an individual (a potential solution) has to the problem which is trying to be optimized. Once it
is not possible to know the nature of that function, the Algorithm 6 generically represents it with multiple dimensions.

\textbf{Algorithm 6. EFM Algorithm}

\begin{algorithm}
\begin{algorithmic}
\Function{EFM}{\( \vec{x}(k) \)}
\State \Return \( f(\vec{x}_0[M](k), \ldots, \vec{x}_1[M](k), \ldots, \vec{x}_{D-1}[M](k)) \)
\EndFunction
\end{algorithmic}
\end{algorithm}

The processing time of this module depends solely on the function \( f(\cdot) \), and as will be shown in the results, it can
consume a lot more clock cycles than any other module. Therefore, the processing time can be specified as

\[ t_{\text{EFM}} = c_{\text{CLK}} \times \frac{1}{\text{CLK}}. \]  
(14)

3.5.4 | Fitness function module

This module is where the evaluation of the entire population is performed by using the problem’s evaluation function.
In order to do that, it utilizes the two previous modules, NFM and EFM, and its operation is described in Algorithm 7.
Essentially, each individual is first normalized (Line 5 of Algorithm 7), later evaluated (Line 6 of Algorithm 7), and finally
a verification is carried out in order to find the best individual in the population (Line 7 of Algorithm 7). In the end, the
index of the best individual, \( j_b \), is returned to be used in the elitism.

\textbf{Algorithm 7. FFM Algorithm}

\begin{algorithm}
\begin{algorithmic}
\Function{FFM}{X(k), y(k)}
\State popsize_t \( j, j_b = 0 \)
\State normalization_t \( \vec{x}(k) \)
\For{\( j \leftarrow 0 \; \text{to} \; N - 1 \)}
\State NFM(\( \vec{x}_j(k), \vec{x}(k) \))
\State \( y_j[B](k) \leftarrow \text{EFM}(\vec{x}(k)) \)
\If{\( y_j[B](k) < y_{j_b}[B](k) \)}
\State \( j_b \leftarrow j \)
\EndIf
\EndFor
\Return \( j_b \)
\EndFunction
\end{algorithmic}
\end{algorithm}

The processing time of this module depends on several operations, including the processing time of the modules NFM and EFM, \( t_{\text{NFM}} \) and \( t_{\text{EFM}} \), previously defined in Equations (13) and (14). Thus, it can be defined as

\[ t_{\text{FFM}} = N \times (c_{\text{for}} + c_{\text{strB}} + c_{\text{strB}} + c_{\text{strB}}) \times \frac{1}{\text{CLK}} + N \times (t_{\text{NFM}} + t_{\text{EFM}}). \]  
(15)

where \( N \) is the population size, CLK is the processor frequency and \( c_{\text{for}}, c_{\text{strB}}, c_{\text{strB}}, \) and \( c_{\text{strB}} \) are, respectively, the number of clock cycles needed for: a loop iteration; a B-bits attribution; a B-bits conditional evaluation; an 8-bits attribution, which is being considered even though it happens only when the condition is true.
3.5.5 | Selection function module

As its name suggests, this module is where the selection step occurs. The selection strategy adopted was the tournament selection with two individuals competing between themselves. With this purpose, the module receives as inputs the fitness values of the entire population of the \( k \)th generation, \( y(k) \), and after randomly choosing two individuals, it returns the index of the winner. In other words, it returns the index of the individual with best fitness value and this procedure is presented in Algorithm 8.

The Lines 2 and 3 of the Algorithm 8 select two random indices of individuals (or chromosomes) from the population. These indices must be between 0 and \( N-1 \), where \( N \) is the size of population. The smaller PRNG function as discussed in Section 3.3 generates an 8-bits number, that is, a number between 0 and 255. Thus, the easiest way to ensure that the produced number remains between 0 and \( N-1 \) is by using the modulus operator, represented by the symbol \( % \). In other words, the index can be calculated as

\[
ich = \text{RNG}() \ % \ (N-1).
\] (16)

However, since the size of the population \( N \) is always a power of 2, the modulus operation, which is computationally costly and needs a significant amount of clock cycles, can be replaced by a bitwise AND operation. This logical operation is represented in Algorithm 8 by the symbol \( \land \), which is faster to a microcontroller to carry out when compared with the modulus operation.

**Algorithm 8. SFM Algorithm**

```plaintext
1: function SFM(y(k))
2:     popsize_t ichx ← RNG-SFM-1() \land (N-1)
3:     popsize_t ichy ← RNG-SFM-2() \land (N-1)
4:     if (y[ichx][M](k) ≤ y[ichy][M](k)) then
5:         return ichx
6:     else
7:         return ichy
8:     end if
9: end function
```

Due to the constraint of the population size \( N \) being less than 256, as described in Section 3.1, both functions RNG-SFM-1() and RNG-SFM-2() were implemented using the function `lfsr_rand8()`), since 8-bits are enough to represent all the possible sizes. Therefore, the processing time of this module can be calculated as

\[
t_{\text{SFM}} = 2 \times (c_{\text{RNG-SFM}}^{\text{CLK}} + c_{\text{atr8}}^{\text{CLK}} + c_{\text{AND}}^{\text{CLK}} + c_{\text{ifB}}^{\text{CLK}}) \times \frac{1}{C_{\text{CLK}}}.
\] (17)

\( t_{\text{SFM}} \) is proportional to: the number of clock cycles needed to generate an 8-bits random number, \( c_{\text{RNG-SFM}}^{\text{CLK}} \); the clock cycles for an 8-bits attribution, \( c_{\text{atr8}}^{\text{CLK}} \); the clock cycles needed for an 8-bits bitwise AND, \( c_{\text{AND}}^{\text{CLK}} \); and the clock cycles needed for an 8-bits conditional operation, \( c_{\text{ifB}}^{\text{CLK}} \). \( C_{\text{CLK}} \) represents the processor frequency.

3.5.6 | Crossover function module

This module performs the crossover operation, where two individuals previously selected are combined together to generate two new individuals, as can be seen in Algorithm 9. The function receives as parameters the index \( i \) from the matrix \( Z(k) \), which represents the new population, so that the newly created two individuals are stored in positions \( i \) and \( i+1 \) from that population. It also receives as parameters the reference for the current population of the \( k \)th generation, \( X(k) \), the reference for the new population, \( Z(k) \) and the indexes of the two individuals from the current population which will be used in the crossover operation, \( ichA \) and \( ichB \).
The crossover is performed using a mask, which is a 8, 16, or 32-bits numerical constant, defined as a macro before the compilation. It is named $\text{mask}[M]$ in Algorithm 9, where $M$ is the number of bits to represent a dimension of an individual and it is defined as

$$\text{mask}[M] = [\text{mask}_0, \text{mask}_1, \ldots, \text{mask}_{M-1}]. \quad (18)$$

Depending on how the mask is configured, it is possible to do different types of crossovers such as one-point, two-point, and uniform. These three possibilities are shown below:

- **One-point**: $\text{mask}_j = 1$ for the interval $j = 0$ to $M/2 - 1$ and $\text{mask}_j = 0$ for the interval $j = M/2$ to $M - 1$;
- **Two-point**: $\text{mask}_j = 1$ for the interval $j = 0$ to $3M/4 - 1$ and $j = 3M/4$ to $M - 1$; and $\text{mask}_j = 0$ for the interval $j = M/4$ to $3M/4 - 1$;
- **Uniform**: $\text{mask}[M]$ defined as desired.

**Algorithm 9. CFM Algorithm**

| 1: function CFM($i, X(k), Z(k), ichA, ichB$) |
| 2: \quad \text{dimensionsize}_t t |
| 3: \quad \text{for} $t \leftarrow 0$ \text{ to } $D - 1$ \text{ do} |
| 4: \quad \quad \text{zi}_t[M](k) \leftarrow (x_{ichA,t}[M](k) \wedge \text{mask}[M]) \vee (x_{ichB,t}[M](k) \wedge \neg \text{mask}[M]) |
| 5: \quad \quad zi_{t+1}[M](k) \leftarrow (x_{ichA,t}[M](k) \wedge \neg \text{mask}[M]) \vee (x_{ichB,t}[M](k) \wedge \text{mask}[M]) |
| 6: \quad \text{end for} |
| 7: \text{end function} |

The processing time of this module is

$$t_{\text{CFM}} = D \times \left[ c_{\text{for CLK}}^{\text{for}} + 2 \times c_{\text{altM CLK}}^{\text{alt}} + c_{\text{ORM CLK}}^{\text{ORM}} + 2 \times c_{\text{ANDM CLK}}^{\text{AND}} + c_{\text{NegM CLK}}^{\text{Neg}} \right] \times \frac{1}{\text{CLK}}. \quad (19)$$

This time is influenced by the number of clock cycles needed for a loop iteration, $c_{\text{forCLK}}^{\text{for}}$, by the clock cycles needed to perform an $M$-bits attribution, $c_{\text{altM CLK}}^{\text{alt}}$, by the clock cycles for an $M$-bits bitwise OR operation, $c_{\text{ORM CLK}}^{\text{ORM}}$, which is represented by the symbol $\vee$, by the clock cycles required for an $M$-bits bitwise AND operation, $c_{\text{ANDM CLK}}^{\text{AND}}$, which is represented by the symbol $\wedge$ and by the clock cycles for necessary for an $M$-bits bitwise NOT operation, $c_{\text{NegM CLK}}^{\text{Neg}}$, represented by the symbol $\neg$. As before, CLK represents the processor frequency.

### 3.5.7 Selection and crossover function module

The purpose of this module is to produce a new offspring to substitute the previous population. For each iteration, it performs a crossover of two individuals previously selected using the two preceding modules, SFM and CFM. As shown in the Algorithm 10, the module receives as inputs: the fitness values of the population, $y(k)$, at the $k$th generation; the current population, $X(k)$; and the new population to be generated, $Z(k)$. It can be noticed that the loop runs only $N/2$ times (the loop uses step 2), since the crossover generates two individuals at every execution.

**Algorithm 10. SCFM Algorithm**

| 1: function SCFM($y(k), X(k), Z(k)$) |
| 2: \quad \text{popsize}_t i |
| 3: \quad \text{for} $i \leftarrow 0$ \text{ to } $N - 1$ \text{ with step } 2 \text{ do} |
| 4: \quad \quad \text{CFM} (i, X(k), Z(k), SFM(y(k)), SMF(y(k))) |
| 5: \quad \text{end for} |
| 6: \text{end function} |
The processing time of this module is defined as

\[ t_{SCFM} = \frac{N}{2} \times \left[ \left( c_{\text{for}}^{\text{CLK}} \times \frac{1}{\text{CLK}} \right) + t_{CFM} + 2 \times t_{SFM} \right], \]  

which depends mainly on the processing time of the modules SFM and CFM, \( t_{CFM} \) and \( t_{SFM} \), and the clock cycles needed for a loop iteration. CLK represents the processor frequency.

### 3.5.8 Mutation function module

This is the module in which the mutation operation occurs as described in Algorithm 11. In this operation, \( P \) individuals mutate as explained in Section 2. The module has as input the matrix that represents the new population, \( Z(k) \), which is filled with individuals generated after the selection and crossover operations. In order to simplify and optimize the mutation, it was defined that the individuals \( z_j[M](k) \) from \( j = 1 \) to \( j = P \) are submitted to this transformation. This is possible since \( P \) can be calculated from the mutation rate, defined in Equation (4).

The chosen strategy for the mutation process was to change one bit in all dimensions of all \( P \) individuals. The position of this bit, \( o \), is randomly chosen by the function RNG-MFM (Line 5 of the Algorithm 11), which is implemented using the function \( \text{lfsr\_rand8()} \) because the highest possible position is 31, since \( M \) must be 8, 16, or 32. Again, for the same reason explained in Section 3.5.5, the modulus operation was replaced by a bitwise AND. Finally, the value modification of the chosen bit is performed with a logical shift and the bitwise XOR operation (Line 7 of the Algorithm 11).

**Algorithm 11. MFM Algorithm**

1. **function** MFM(Z(k))
2. \( \text{popsize}_t \ j \)
3. \( \text{dimensionsize}_t \ i \)
4. **for** \( j \leftarrow 1 \) to \( P \) **do**
5. \( \text{chromosomesize}_t \ o \leftarrow \text{RNG-MFM}() \wedge (M - 1) \)
6. **for** \( i \leftarrow 0 \) to \( D - 1 \) **do**
7. \( z_{j,i}[M](k) \leftarrow z_{j,i}[M](k) \oplus (1 \ll o) \)
8. **end for**
9. **end for**
10. **end function**

The calculation of the processing time of this module is presented as

\[ t_{MFM} = P \times \left[ c_{\text{for}}^{\text{CLK}} + c_{\text{atrs}}^{\text{CLK}} + c_{\text{RNG-MFM}}^{\text{CLK}} + c_{\text{AND8}}^{\text{CLK}} + D \times (c_{\text{for}}^{\text{CLK}} + c_{\text{atrs}}^{\text{CLK}} + c_{\text{XOR}}^{\text{CLK}} + c_{\text{ShiftM}}^{\text{CLK}}) \right] \times \frac{1}{\text{CLK}}, \]  

where \( P \) is the number of mutated individuals, \( D \) is the number of dimensions of the individual and CLK is the processor clock. In addition, the following variables represent that amount of clocks needed to: a loop iteration \( (c_{\text{for}}^{\text{CLK}}) \); an 8-bits attribution \( (c_{\text{atrs}}^{\text{CLK}}) \); generate an 8-bits random number \( (c_{\text{RNG-MFM}}^{\text{CLK}}) \); to carry out an 8-bits bitwise AND \( (c_{\text{AND8}}^{\text{CLK}}) \); to a \( M \)-bits attribution \( (c_{\text{ShiftM}}^{\text{CLK}}) \); to carry out a \( M \)-bits bitwise XOR operation \( (c_{\text{XOR}}^{\text{CLK}}) \); and to a \( M \)-bits logical shift \( (c_{\text{ShiftM}}^{\text{CLK}}) \).

### 3.5.9 Update function module

This module represents the last stage of the GA. It is in this module that the old population of the \( k \)th generation, \( \mathbf{X}(k) \), is updated with new individuals stored in \( Z(k) \) in order to create the population of the \( k + 1 \) generation, \( \mathbf{X}(k + 1) \). This procedure is presented in Algorithm 12.

In addition to the references for the matrices \( \mathbf{X}(k) \) and \( Z(k) \), this module takes also as input the index of the best individual of the current generation, \( j_b \). This last input is used by the elitism feature, where the individual \( x_{j_b,k}[M](k) \) is
forwarded directly to the first position of the population of next generation, as can be seen in Line 5 of the Algorithm 12. In order to complete the population of the \( k+1 \) generation the new individuals stored in \( Z(k) \) are copied to the matrix \( X(k) \).

**Algorithm 12. UFM Algorithm**

```plaintext
1: function UFM(jb, X(k), Z(k))
2:   popsize_t j
3:   dimensionsize_t i
4:   for i ← 0 to D − 1 do
5:     \( x_0,i \rightarrow j, [M](k) \leftarrow x_{jb,i} [M](k) \)
6:   end for
7:   for j ← 1 to N − 1 do
8:     for i ← 0 to D − 1 do
9:       \( x_{j,i} [M](k) \leftarrow z_{j,i} [M](k) \)
10:   end for
11: end for
12: end function
```

The processing time of this module is calculated as

\[
t_{\text{UFM}} = \left[ D \times (c_{\text{for}} + c_{\text{atM} \cdot \text{CLK}}) + N \times (c_{\text{for}} + D \times (c_{\text{for}} + c_{\text{atM} \cdot \text{CLK}})) \right] \times \frac{1}{\text{CLK}}. \tag{22}
\]

\( t_{\text{UFM}} \) is proportional to: the size of the population, \( N \); the number of dimensions of the individuals (or chromosomes), \( D \); and the clock cycles for a loop iteration, \( c_{\text{for} \cdot \text{CLK}} \); the clocks cycles for a \( M \)-bits attribution, \( c_{\text{atM} \cdot \text{CLK}} \). CLK is the clock of the processor.

### 3.5.10 New population function module

This is the last module proposed in this work and its purpose is to encapsulate the three previous modules, SCFM, MFM, and UFM. It is also here that the life cycle of the matrix \( Z[M](k) \) occurs, which stores temporarily the individuals for the new population. In other words, it is where that matrix is allocated in memory and later freed. The structure of this module is presented in Algorithm 13 and its inputs are: a reference to the vector \( y(k) \), which stores the fitnesses of the individuals; a reference to the matrix \( X[M](k) \), which stores the current population and the index of the best individual of the current generation, \( jb(k) \).

**Algorithm 13. NPFM Algorithm**

```plaintext
1: function NPFM(y(k), X[M](k), jb(k))
2:   chromosome_t Z[M](k)
3:   SCFM(y(k), X(k), Z(k))
4:   MFM(Z(k))
5:   UFM(X(k), Z(k), jb(k))
6: end function
```

The processing time consists in the sum of the times spent by the modules SCFM, MFM, and UFM, that is, \( t_{\text{SCFM}}, t_{\text{MFM}}, \) and \( t_{\text{UFM}} \). The expressions for calculating each one of these was previously defined. The final expression can be calculated as

\[
t_{\text{NPFM}} = t_{\text{SCFM}} + t_{\text{MFM}} + t_{\text{UFM}}. \tag{23}
\]
4 | RESULTS

In order to validate the GA implementation proposed in Section 3 and evaluate its operation as well as its performance and resources consumption, a program was developed using the C programming language targeting Atmel microcontrollers. The development was performed using the software Atmel Studio 7, provided by Atmel. The code strictly followed the algorithms presented for each module, so that each one was defined as a C function. In addition, programming recommendations provided by Atmel had also been taken into account in order to generate a satisfactory code. The source code used in this project to generate the results can be accessed in this public repository: https://github.com/DenisMedeiros/EmbeddedGeneticAlgorithms.

The microcontroller used to validate this implementation was the ATmega328P, which uses AVR architecture. According to the manufacturer, the chip is an 8-bits µC and has the potential of 1 millions instructions per second (MIPS) per MHz and can reach a peak of 20 MIPS running with a clock of 20 MHz. Moreover, it has 32 KB of program memory and 2 KB of data memory.

Even though µCs have become robust, many of them running 32-bits instructions in modern architectures such as ARM and with more program and data memories, the choice of the Atmega328p was intentional. The aim is to use the GA proposed in this work on a simple and light microcontroller, which usually has lower price and consumes less power. This is more suitable for many applications, such as those applied in the IoT, for example. Once it is validated in a simple and limited platform such as this µC, it can be configured to run in more complex ones as well.

It is necessary to mention that the GA was implemented as an embedded system using the development kit Arduino Uno, which uses the ATmega328P. Besides the microcontroller, the Arduino platform comes with a built-in programming interface to write the program into the flash memory and with other interfaces which help the development and test of the system.

Finally, all the charts presented in this work were generated using the Python library Matplotlib.

4.1 | Resources consumption

The data collection for the information presented in this section was performed in the following way:

- For program and data memory consumption, it was used reports generated by the Atmel Studio 7 during the compilation and during the upload of the program into the microcontroller;
- For processing time, it was used the Atmel Studio 7 which can measure the clock cycles during the debugging as well as oscilloscope observations while the µC was running the code.

For all measurements, the system was compiled using the optimization flag -O2, which makes the compiler optimize its processing time. On the other hand, the user can change this flag to -Os in order to optimize the size of the program, although, as will be shown later, the program size was already small enough.

4.1.1 | Memory consumption

The first results were the program and data memory consumption. The program memory is the nonvolatile memory where the compiled program is stored to be executed by the microcontroller. The data memory is the dynamic and volatile memory where the variables are stored during when the program is running. The way how this data memory is used depends on how the variables are defined:

- Static memory: the memory consumed by global and static variables and it is allocated during the whole program execution. That means this section of the memory cannot be freed and used by other variables.
- Stack memory: the memory used by local variables and that can be allocated and freed according to the lifetime of those variables (e.g., a local variable defined inside of a function will be freed after the function is finished).

After the program compilation, the Atmel Studio 7 reports the program memory necessary to store it as well as the initial data memory used by global and static variables, that is, the static memory. Nevertheless, to measure the maximum
stack memory consumed by the program during its execution (the peak of consumption), it was used a different methodology since this memory is used in a dynamic way. Since it is known that the word size of the ATmega328P architecture is 1 byte, then the maximum data memory consumption can be calculated as follows:

1. Find the start and end address of the stack in memory;
2. Before the execution of the program, mark all stack addresses with a default mask (0xC5, for example);
3. Carry out the program;
4. Count the amount of bytes which are still with that mask and, based on this, calculate the maximum number of bytes used during the running.

Therefore, the program and data memory consumption are presented in Tables 1 to 3 for different configurations of the GA. In order to simplify the measurements, all simulations were performed with a fixed number of generations \( K = 64 \), since this affects only the processing time. In addition, the evaluation function used was \( f_1(x) = (x - 2) \times (x - 4) \), with dimension \( D = 1 \), to avoid the use of external libraries, such as \texttt{libm}, which may be necessary for some mathematical operations such as trigonometric functions. Finally, the crossover was configured as one-point and the number of mutated individuals was \( P = 1 \).

As presented in Table 1, the program memory was significantly low. The compiled code averaged about 3.5 KB. This represents approximately 11% of the ATmega328P capacity and was considerably smaller than the results presented by Reference 14, where their implementation consumed, in the best case, 78.4 KB.

Regarding Table 2, the initial data memory consumption was almost null. This can be explained by the avoidance of the use of global and static variables, since the memory used by them is retained during all execution, even if these variables are barely used. Thus, in this work the whole data memory is available to be used as stack.

The most important data memory consumption analysis is presented in Table 3, since it shows how much it is consumed from the stack after executing the GA. In the simplest case, with 8 individuals represented in 8-bits, the program consumed 143 bytes or about 7% of the data memory of the ATmega328P. In a practical situation with 64 individuals represented in 16-bits, it was consumed 558 bytes or about 27% of the memory. Finally, with 128 individuals represented in 32-bits, about 80% of the whole memory was consumed.

### Table 1
Program memory consumption (in bytes), of a total of 32 KB

| Individual size (M) | Population size (N) |
|---------------------|---------------------|
| 16                  | 32                  | 64                  | 128                 |
| 8-bits              | 3212                | 3212                | 3216                | 3216                |
| 16-bits             | 3412                | 3416                | 3416                | 3410                |
| 32-bits             | 3678                | 3678                | 3670                | 3670                |

### Table 2
Static memory consumption (in bytes), of a total of 2 KB

| Individual size (M) | Population size (N) |
|---------------------|---------------------|
| 16                  | 32                  | 64                  | 128                 |
| 8-bits              | 4                   | 4                   | 4                   | 4                   |
| 16-bits             | 4                   | 4                   | 4                   | 4                   |
| 32-bits             | 8                   | 8                   | 8                   | 8                   |

### Table 3
Stack memory consumption (in bytes), of a total of 2 KB

| Individual size (M) | Population size (N) |
|---------------------|---------------------|
| 16                  | 32                  | 64                  | 128                 |
| 8-bits              | 143                 | 239                 | 431                 | 815                 |
| 16-bits             | 174                 | 302                 | 558                 | 1070                |
| 32-bits             | 242                 | 434                 | 818                 | 1586                |
Looking at these results, both the sizes of population and individuals increase the consumption of data memory, but increasing \( N \) is more memory consuming, since its growth is almost linear. Furthermore, it can be inferred that the number of dimensions \( D \) must produces a growth on the memory consumption analogous to \( N \), because it is equivalent to multiply the population size by \( D \).

Therefore, it is important to choose the correct GA configuration and take into consideration the memory limitations of the microcontroller. The parameter that affects the most the memory consumption is the population size \( N \). In addition, most applications will require more or less precision, which affects the size of \( M \). Hence, both parameters must be adjusted according to the problem requirements in order to make the GA implementation feasible.

### 4.1.2 Processing time

The first part of the processing time results was collected from the reports generated by the Atmel Studio 7 during the debugging process. The objective was to analyze the number of clock cycles consumed by each module from this implementation in order to identify critical parts of the program, that is, which sections are more costly. With the purpose of calculating the processing time, \( t \), in seconds, one must divide the number of clock cycles, \( c_{CLK} \), by the processor clock, which in the Arduino Uno development board is 16 MHz. Hence, this is calculated as

\[
t(s) = \frac{c_{CLK}}{16 \times 10^6}.
\]  

(24)

The first measurements were performed with the simplest possible configuration of the GA: the population size was \( N = 2 \); number of generations \( K = 1 \); number of dimensions \( D = 1 \); number of mutated individuals (or chromosomes) \( P = 1 \); the EFM was set as the same function used in the previous section, that is, \( f_1(x) \). Moreover, it was not considered any processing time of a module inside another (such as SFM and CFM inside of NPFM, for instance), since each one was measured separately. Finally, the only changed parameter was the individual size, \( M \), since it affects the PRNG.

Table 4 displays the clock cycles and processing time consumed by each module, respectively. Although the EFM used a simple function, it was the module responsible for the longest processing time. This can be justified by the fact that the evaluation function makes mathematical operations using float point numbers. This explains the reason why the NFM also spends numerous clock cycles, as it needs to evaluate the expression shown in Equation (12) with float point numbers as well. Finally, other modules that also stand out are the IFM and SFM, due to their need to use the PRNG, as explained in Algorithms 4 and 8.

| Module       | Individual size \((M)\) | 8-bits | 16-bits | 32-bits |
|--------------|-------------------------|--------|---------|---------|
|              | \( c_{CLK} \) | \( t(\mu s) \) | \( c_{CLK} \) | \( t(\mu s) \) | \( c_{CLK} \) | \( t(\mu s) \) |
| IFM          | 56 | 3.50 | 74 | 4.62 | 128 | 8.00 |
| NFM          | 305 | 19.0 | 297 | 18.6 | 353 | 22.1 |
| EFM          | 386 | 24.1 | 379 | 23.7 | 373 | 23.31 |
| FFM          | 222 | 13.9 | 252 | 15.8 | 253 | 15.81 |
| SFM          | 142 | 8.88 | 146 | 9.12 | 141 | 8.81 |
| CFM          | 59 | 3.69 | 74 | 4.62 | 127 | 7.94 |
| SCFM         | 13 | 0.812 | 31 | 1.94 | 36 | 2.25 |
| MFM          | 63 | 3.94 | 72 | 4.50 | 84 | 5.25 |
| UFM          | 29 | 1.81 | 40 | 2.50 | 65 | 4.06 |
| NPFM         | 37 | 2.31 | 42 | 2.63 | 78 | 4.88 |

**Table 4** Clock cycles consumed by each module of the GA

Abbreviations: CFM, crossover function module; EFM, evaluation function module; FFM, fitness function module; GA, genetic algorithm; IFM, initialization function module; MFM, mutation function module; NFM, normalization function module; NPFM, new population function module; SCFM, selection and crossover function module; SFM, selection function module; UFM, update function module.
**Table 5** Clock cycles consumed by the entire GA

| Individual Size (M) | 8-bits | 16-bits | 32-bits |
|---------------------|--------|---------|---------|
|                     | $c_{CLK}$ | $t(\mu s)$ | $c_{CLK}$ | $t(\mu s)$ | $c_{CLK}$ | $t(\mu s)$ |
|                     | 3766   | 235     | 3845    | 240     | 4202    | 263     |

Abbreviation: GA, genetic algorithm.

**Table 6** Clock cycles required by other evaluation functions (or fitness functions)

| $f(\overline{x})$ | Chromosome size (M) | 8-bits | 16-bits | 32-bits |
|-------------------|----------------------|--------|---------|---------|
|                   |                      | $c_{CLK}$ | $t(\mu s)$ | $c_{CLK}$ | $t(\mu s)$ | $c_{CLK}$ | $t(\mu s)$ |
| $f_2(\overline{x})$ | 392                 | 24.5    | 366    | 23.8    | 365    | 22.8    |
| $f_3(\overline{x})$ | 428                 | 26.8    | 441    | 27.6    | 433    | 27.1    |
| $f_4(\overline{x})$ | 4259               | 266    | 4409    | 276    | 4377    | 274    |
| $f_5(\overline{x})$ | 4824               | 302    | 4960    | 310    | 4717    | 294    |

The processing time of the entire GA, presented in Table 5, is greater than the sum of the times of all above modules. This can be explained by the existence of an extra overhead while invoking each function and also caused by the memory allocation of local variables in the stack. Furthermore, some modules are executed more than once, such as EFM and NFM, as they are applied to the whole population.

In addition to the previous result, it was also measured the number of clock cycles for other evaluations functions, which are commonly used in the literature to evaluate metaheuristics. The purpose is to show how costly these functions can be in comparison to the entire GA, even in a simple simulation as performed before. These functions are listed below and it is important to mention that only $f_4(\overline{x})$ has one dimension while $f_2(\overline{x})$, $f_3(\overline{x})$, and $f_5(\overline{x})$ have two. The reason to choose these functions is that they are usually used to test metaheuristics and optimization algorithms.

- $f_2(\overline{x}) = \overline{x}_0^2 + \overline{x}_1^2$
- $f_3(\overline{x}) = 2\overline{x}_0 + 3\overline{x}_1 + 5$
- $f_4(\overline{x}) = 21.5 + \overline{x}_0(\sin(40\pi \overline{x}_0) + \cos(20\pi \overline{x}_0))$
- $f_5(\overline{x}) = 10 + \overline{x}_0^2 - 10 \cos(2\pi \overline{x}_0) + \overline{x}_1^2 - 10 \cos(2\pi \overline{x}_1)$

Table 6 shows the number of clock cycles and time required by each evaluation function (or fitness function). The execution of either of these last two functions, regardless of having one or two dimensions, consume more processing time that the whole GA, as shown in Table 5. In fact, for most metaheuristics the evaluation function is the most critical part. In these particular examples, the need of calculating trigonometric functions was sufficient to make them overly costly.

In order to validate the time estimations based on the number of clock cycles, an oscilloscope was used to measure the real processing time. Hence, the implementation was slightly modified so that for each execution of the GA or of some specific part of the program, the microcontroller generated an oscillating digital output signal with low and high values on a GPIO pin. In other words, it generated a square wave in which the half of the period represented the total time needed to execute that part of the program. This idea is presented in Algorithm 14.

**Algorithm 14.** Algorithm used to generate the signal measured by oscilloscope

1. **while** true **do**
2.   Execute the GA or part of the program.
3.   Toggle the output of a specific GPIO pin.
4. **end while**
Tables 7 and 8 show the measured time for the entire GA and for the evaluation functions presented before, respectively. They show the processing time, $t$, in microseconds, and the relative error comparing to the estimations using clock cycles and the Equation (24). Looking at the presented values, the relative error is low for almost all cases, even when considering the oscilloscope imprecision and the signal toggle settling time. Therefore, this proves that those estimations are fair and reasonable.

In order to conclude the processing time investigation, a few simulations were conducted with the purpose of understanding the time complexity of the implemented GA, according to the parameters configuration. For these results, the analyzed parameters were the population size, $N$, and the number of generations, $K$. The individuals were set with number of dimensions $D = 1$, represented in $M = 16$ bits; the number of mutated individuals was fixed as $P = 2$; and the evaluation function was kept as $f_1(\mathbf{x})$. These results were obtained using the Atmel Studio 7 and are shown in Table 9.

To illustrate the results shown in Table 9, the data was plotted on a chart in Figures 3 and 4. The points represent the measured values and the dashed lines represent the best polynomial approximation. Therefore, as for the growth of the population size $N$ and as for the growth of the number of generations $K$ the processing time complexity was observed approximately linear.

From the processing time results, it can be inferred that all modules are optimized and the EFM, which represents the evaluation function and can be quite complex, is the critical part of the GA. In addition, the NFM must be considered a secondary critical part, since it performs mathematical operations with float point numbers and it is applied to the entire population in all generations, similarly to the EFM. Specifically, both EFM and NFM are executed $K \times N$ times during the GA. One option to reduce the processing time of those modules would be to use fixed-point arithmetic or to spend more memory to use lookup tables instead to represent both functions or, at least, parts of them.

Finally, in most simulations the whole GA could be executed in few milliseconds. This makes it feasible to be used in several problems such as real-time applications or in areas such as robotics, industrial automation, automotive applications, and so on. However, to make this possible, the GA needs to be properly tuned regarding its parameters so that it can meet the time requirements and memory limitations.

### Table 7

| Individual size ($M$) | 8-bits | 16-bits | 32-bits |
|-----------------------|--------|---------|---------|
| $t(\mu s)$ | Error (%) | $t(\mu s)$ | Error (%) | $t(\mu s)$ | Error (%) |
| 240 | 1.75 | 248 | 3.20 | 270 | 2.81 |

Abbreviation: GA, genetic algorithm.

### Table 8

| Individual size ($M$) | 8-bits | 16-bits | 32-bits |
|-----------------------|--------|---------|---------|
| $f(\mathbf{x})$ | $t(\mu s)$ | Error (%) | $t(\mu s)$ | Error (%) | $t(\mu s)$ | Error (%) |
| $f_2(\mathbf{x})$ | 24.8 | 1.22 | 23.2 | 1.42 | 23.6 | 3.45 |
| $f_3(\mathbf{x})$ | 27.9 | 4.49 | 28.2 | 2.13 | 26.9 | 0.60 |
| $f_4(\mathbf{x})$ | 270 | 1.43 | 278 | 0.88 | 276 | 0.89 |
| $f_5(\mathbf{x})$ | 306 | 1.49 | 314 | 1.29 | 298 | 1.08 |

### Table 9

| Number of generations ($K$) | Population size ($N$) |
|-----------------------------|-----------------------|
| 16 | 32 | 64 | 128 |
| 16 | 16.5 | 32.2 | 65.0 | 129 |
| 32 | 32.0 | 62.7 | 126 | 252 |
| 64 | 63.2 | 123 | 249 | 496 |
| 128 | 126 | 246 | 494 | 986 |

Abbreviation: GA, genetic algorithm.
4.2 Validation with HIL

In order to verify whether the GA was working as expected, experiments using the Hardware-in-the-loop technique were performed. In this model, the microcontroller is linked to the computer so that the program is executed on the μC and they can share data, such as parameter values and results. During the experiments, the communication between both devices used the universal asynchronous receiver/transmitter (USART) interface together with a Python program which was used to receive data from the μC and to generate the plots.

The first experiment consisted in finding the global minimum of the function $f_2(x)$ and the second one focused on finding a local maximum, between 0.8 and 1.0, of $f_4(x)$. The charts of both functions are presented in Figures 5 and 6, respectively. Looking at them, it can be noticed that the global minimum of $f_2(x)$ is in the point $(x_0 = 0, x_1 = 0)$ and the local maximum of $f_4(x)$ is located around $x = 0.910$.

For the function $f_2(x)$, the GA was configured as follows: $N = 32$ individuals, $D = 2$ dimensions, $M = 16$ bits, number of generations $K = 32$, $P = 2$ mutated individuals and normalization limits $L_{\min} = -4.0$ and $L_{\max} = 4.0$. The results of this experiment are shown in Figures 7 and 8. Each dimension converged at different moments to the expected result, but $x_0$ needed slightly more iterations to accomplish it.
**Figure 5** Chart of $f_2(\mathbf{x})$, with $x_0$ and $x_1$ from -5 to 5

**Figure 6** Chart of $f_4(\mathbf{x})$, with $x$ from 0.8 to 1.0

**Figure 7** HIL result for coordinate $x_0$ of $f_2(\mathbf{x})$. HIL, hardware-in-loop
For the function $f_4(x)$, the GA was configured as follows: $N = 16$ individuals, $D = 1$ dimensions, $M = 16$ bits, number of generations $K = 32$, $P = 2$ mutated individuals and normalization limits $L_{\text{min}} = 0.8$ and $L_{\text{max}} = 1.0$. The result for this function is presented in Figure 9. As can be seen in it, after few generations the GA converged to the correct result.

### 4.3 Comparison with other implementation

As was stated in the introduction, there are only few works in the literature with a similar objective to the one presented in this work, that is, to propose a GA implementation targeting 8-bits microcontrollers. In addition, the majority of those works did not present detailed results about memory consumption and processing time nor explained how the GA was implemented. Thus, the only comparison that could be conducted was with the implementation proposed by Reference 12, since the source code of the implementation was provided.

The implementation presented by Reference 12 used the C++ programing language and the Arduino IDE. The author used several libraries provided by the development platform to implement his project. This publication did not
present results about memory consumption and processing time, but the author pointed that his implementation had the following constraints:

- The population size is less than 100;
- The individuals are represented in $M = 32$-bits;
- The fitness value is represented between 0 and 100.

In order to validate that implementation, the source code had to be slightly changed in order to be compiled and debugged on Atmel Studio 7. A function `main`, which is not required by the Arduino IDE, had to be created with the purpose to call the functions `setup` and `loop` that exist by default in programs developed using that tool. Moreover, the population size, $N$, and the number of generations, $K$, were configured to 64 so a fair comparison with the present work could be performed. Regarding the mutation operation, the compared work used a mutation rate of 0.001 and the original goal is to maximize the function $f(x) = x$, where $x$ is a 32-bits unsigned integer number. With this purpose of making this evaluation, the respective function from Reference 12 counted the amount of bits with value 1 in an individual, which means that the one with the most bits 1 is considered the best. Finally, all the commands from the analyzed source code that generated reports via USART, such as in the piece containing `ga.reportStatistics` and `ga` were disabled in an effort to produce fair results.

After making these adjustments, both implementations were configured with the exact same parameters and several simulations using the Atmel Studio 7 were performed for the comparison of both. In order to illustrate that the present implementation was able to maximize the target function, an experiment using the HIL approach was performed and the result is shown in Figure 10. As can be seen, after approximately 50 generations the GA converged close to the correct answer, which is $2^{32}$ or 4,294,967,296.

The results for both implementations are presented in Table 10. The results were obtained for $M = 32$, $N = 64$, and $K = 64$ on ATmega328P with 16 MHz. As depicted, the implementation here proposed achieved a better performance in all aspects. The work presented by Reference 12 spent more memory and processing time because it used object-oriented programming for implementing the GA, which produced an extra overhead due the abstraction provided by it. Furthermore, that implementation had numerous global variables, which retained data program from the beginning of the execution. Hence, it is evidenced that the implementation proposed in the present work is well optimized and more efficient than other found in the literature. The memory reduction was about $\approx 1.11$, $\approx 237.67$, and $\approx 2.46$ for program memory, static memory, and stack memory, respectively. The speedup about the processing time was about $\approx 17.73$. 

![Figure 10](image-url)
TABLE 10 Comparison between the proposed implementation (GA1) and the work presented in Reference 12 (GA2)

|                      | This work proposal - GA1 | GA2 | Memory reduction |
|----------------------|--------------------------|-----|------------------|
| Program memory (bytes) | 4044                     | 4514| $\approx 1.11 \times$ |
| Static memory (bytes)  | 6                        | 1426| $\approx 237.67 \times$ |
| Stack memory (bytes)   | 620                      | 1527| $\approx 2.46 \times$ |
| Speedup               |                          |     |                  |
| Processing time (s)    | 0.15                     | 2.66| $\approx 17.73 \times$ |

Note: The results were obtained for $M = 32$, $N = 64$ and $K = 64$ on ATmega328P with 16 MHz.

TABLE 11 Speedup comparison between the proposed implementation (GA1) and the work presented in Reference 12 (GA2)

| Clock frequency (MHz) | Processing time (s) | Speedup     |
|-----------------------|---------------------|-------------|
| 4                     | 0.60                | $\approx 4.43 \times$ |
| 8                     | 0.30                | $\approx 8.86 \times$ |
| 16                    | 0.15                | $\approx 17.73 \times$ |

Note: The results were obtained for $M = 32$, $N = 64$ and $K = 64$ on ATmega328P.

TABLE 12 Power-saving comparison between the proposed implementation (GA1) and the work presented in Reference 12 (GA2)

| Clock frequency (MHz) | $V_{cc}$ (V) | $I_{cc}$ (mA) | Power consumption (mW) | Power-saving |
|-----------------------|--------------|---------------|------------------------|-------------|
| 4                     | 3            | 1.5           | 6                      | $\approx 7.67 \times$ |
| 8                     | 5            | 5.2           | 26                     | $\approx 1.77 \times$ |
| 16                    | 5            | 9.2           | 46                     | 1 $\times$    |

Note: The results were obtained for $M = 32$, $N = 64$ and $K = 64$ on ATmega328P.

Based in Equation (24) and Tables 10, 11 shows the processing time for several clock frequency can be used on ATmega328P microcontroller. The speedup were $\frac{2.66}{0.60} \approx 4.43 \times$ and $\frac{2.66}{0.30} \approx 48.86 \times$ for 4 and 8 MHz, respectively. Even for small clock frequency, the results show speedup gains over the GA2 proposal.

Table 12 shows safe operating voltage, $V_{cc}$ and the typical current consumption, $I_{cc}$, used with different operating clock frequency on ATmega328P microcontroller. This values was obtained from ATmega328P datasheet. In additional, Table 12 shows the power consumption and the power-saving for several values of clock, $V_{cc}$ and $I_{cc}$. The power-saving was calculated regards the proposal GA2 and the power-saving was about $\frac{46}{6} \approx 7.67 \times$ and $\frac{46}{26} \approx 1.77 \times$ for 4 MHz and 8 MHz, respectively. That means this implementation can be configured to work with lower clock speed and voltage, reduce drastically the power consumption, and still have good performance when compared with other implementation in the literature.

5 | CONCLUSIONS

This work presented an implementation proposal of GAs targeting low-power, low-cost, and low-size-memory devices, such as microcontrollers, which is optimized in terms of memory consumption and processing time. The details concerning the implementation and the strategies used to make the algorithm faster and more compact, such as the parameters constraints and the generation of pseudorandom numbers, were provided. In the end, it was also presented the results for the µC ATmega328P with respect to resources consumption, a validation of the GA using the HIL approach and a comparison with another implementation from the literature, which demonstrated this work to have a superior optimization. Hence, it can be concluded that this implementation has proven its feasibility to run on microcontrollers, even in applications where the time constraints are in the order of hundreds of milliseconds. Thus, the GA proposed in this work has potential to be applied in different types of applications such as IoT, for
example. However, as discussed in the results, the GA parameters need to be well configured according to each particular problem so the implementation can moderately consume data memory and the processing time can meet the defined requirements.

ACKNOWLEDGEMENTS
This study was partially financed by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)—Finance Code 001. The authors wish to acknowledge the financial support provided by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).

PEER REVIEW INFORMATION
Engineering Reports thanks the anonymous reviewers for their contribution to the peer review of this work.

CONFLICT OF INTEREST
Authors have no conflict of interest relevant to this article.

AUTHOR CONTRIBUTIONS
Denis Medeiros: contributed to conceptualization; formal analysis; methodology; validation; writing-original draft; writing-review and editing. Matheus Torquato: contributed to conceptualization; data curation; formal analysis; investigation; methodology; validation; writing-original draft; writing-review and editing. Marcelo Fernandes: contributed to conceptualization; formal analysis; funding acquisition; investigation; methodology; resources; supervision; validation; writing-original draft; writing-review and editing.

DATA AVAILABILITY STATEMENT
The data that supports the findings of this study, the complete source code, is openly available in GitHub at https://github.com/DenisMedeiros/EmbeddedGeneticAlgorithms.

ORCID
Marcelo A. C. Fernandes https://orcid.org/0000-0001-7536-2506

REFERENCES
1. Qian K, Den Haring D, Cao L. Survey of Popular Microcontrollers. Boston, MA: Springer; 2009:203-221.
2. Gaikwad PP, Gabhane JP, Golait SS. A survey based on Smart Homes system using Internet-of-things. 2015: 0330-0335
3. Sawulski J, Lawryczuk M. Real-time optimisation of control strategy for a low fuel consumption vehicle engine using STM32 microcontroller: preliminary results. 2019: 4246-4251
4. Al-Kofahi MM, Al-Shorman MY, Al-Kofahi OM. Toward energy efficient microcontrollers and Internet-of-things systems. Comput Electr Eng. 2019;79:106457. https://doi.org/10.1016/j.compeleceng.2019.106457.
5. Situmorang Z, Situmorang IA. Intelligent fuzzy controller for a solar energy wood dry kiln process. 2015: 152-157
6. Ortega-Zamorano F, Jerez JM, Muñoz DU, Luque-Baena RM, Franco L. Efficient implementation of the backpropagation algorithm in FPGAs and microcontrollers. IEEE Trans Neural Netw Learn Syst. 2016;27(9):1840-1850. https://doi.org/10.1109/TNNLS.2015.2460991.
7. Dong H, Liu Y, Yang C, Ao M. Coherent accumulation by using sequence shifting and genetic algorithm. Optik. 2017;138:137-144. https://doi.org/10.1016/j.ijleo.2017.03.080.
8. Eldahab YEA, Saad NH, Zekry A. Enhancing the tracking techniques for the global maximum power point under partial shading conditions. Renew Sustain Energy Rev. 2017;73:1173-1183. https://doi.org/10.1016/j.rser.2017.02.029.
9. Shelokar P, Kulkarni A, Jayaraman VK, Srarry P. Metaheuristics in Process Engineering: A Historical Perspective. Cham: Springer International Publishing; 2014:1-38.
10. Holland JH. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Boston, MA: MIT Press; 1992.
11. Garzia F, Veca GM. Evolutionary computation and genetic algorithms for energy management and conservation. 2002: 386-393
12. Alves N. Implementing genetic algorithms on arduino micro-controllers. 2010; arXiv preprint arXiv:1002.2012.
13. Mamdoohi G, Abas AF, Samsudin K, Ibrahim NH, Hidayat A, Mahdi MA. Implementation of genetic algorithm in an embedded microcontroller-based polarization control system. Eng Appl Artif Intell. 2012;25(4):869-873. Special Section: Dependable System Modelling and Analysis. https://doi.org/10.1016/j.engappai.2012.01.018.
14. Dendaluce M, Valera JJ, Gómez-Garay V, Irigoyen E, Larzabal E. Microcontroller implementation of a multi objective genetic algorithm for real-time intelligent control. In: Herrero Â, Baruque B, Klett F, et al., eds. International Joint Conference SOC’13-CISIS’13-ICEUTE’13. Switzerland: Springer International Publishing, Cham; 2014:71-80.
15. Torquato MF, Fernandes MAC. High-performance parallel implementation of genetic algorithm on FPGA. Circuits Syst Signal Process. 2019;38(9):4014-4039. https://doi.org/10.1007/s00034-019-01037-w.

16. Yordzhev KY. An example for the use of bitwise operations in programming. 2012; arXiv preprint arXiv:1201.1468.

17. Yordzhev K. The bitwise operations related to a fast sorting algorithm. 2013; arXiv preprint arXiv:1312.0138.

18. Kim M, Smaradgis P. Bitwise neural networks. 2016; arXiv preprint arXiv:1601.06071.

19. Cislak A, Grabowski S. Lightweight fingerprints for fast approximate keyword matching using bitwise operations. 2017; arXiv preprint arXiv:1711.08475.

20. dAVF O, MAC F. Dynamic planning navigation strategy for mobile terrestrial robots. Robotica. 2016;34(3):568-583. https://doi.org/10.1017/S0263574714001702.

21. Silva CA, De Oliveira ÁV, Fernandes MA. Validation of a dynamic planning navigation strategy applied to mobile terrestrial robots. Sensors. 2018;18(12):4322.

22. Lamini C, Benhlima S, Elbekri A. Genetic algorithm based approach for autonomous mobile robot path planning. Procedia Comput Sci. 2018;127:C:180-189.

23. Millo Federico, Arya Pranav, Mallamo Fabio. Optimization of automotive diesel engine calibration using genetic algorithm techniques. Energy. 2018;158:807–819. http://dx.doi.org/10.1016/j.energy.2018.06.044.

24. Kalaivanan AP, Sakthivel G. Self tuning genetic algorithm to achieve maximum thermal efficiency by monitoring combustion characteristics with vibration/acoustic sensors. Tech. Rep., SAE Technical Paper; 2017.

25. Talbi EG. Metaheuristics: From Design to Implementation. 1st ed. Hoboken, NJ: John Wiley & Sons; 2009.

26. Razali NM, Geraghty J. Genetic algorithm performance with different selection strategies in solving TSP. Paper presented at: International Association of Engineers Hong Kong; Vol. 2; 2011:1-6.

27. Luke S. Essentials of Metaheuristics. Vol 2. Fairfax, Virginia: Lulu Raleigh; 2013.

28. Gendreau M, Potvin JY. Handbook of Metaheuristics. Vol 2. 2nd ed. Boston, MA: Springer; 2010.

29. Free Software Foundation. The GNU C Library. https://www.gnu.org/software/libc/manual/; 2018.

30. Steele GL Jr, Lea D, Flood CH. Fast Splittable Pseudorandom Number Generators. Vol 49. New York, NY: ACM; 2014:453-472.

31. Microchip. Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR MicrocontrollersThe C Preprocessor. http://ww1.microchip.com/downloads/en/AppNotes/doc8453.pdf; 2018.

32. Hissoiny S, Després P, Ozell B. Using graphics processing units to generate random numbers. arXiv preprint arXiv:1101.1846 2011.

33. Hunter JD. Matplotlib: a 2D graphics environment. Comput Sci Eng. 2007;9(3):90-95. https://doi.org/10.1109/MCSE.2007.55.

AUTOR BIOGRAPHIES

Denis R. da S. Medeiros received the B.Sc. degree in Science and Technology in 2017 and the B.E. degree in Computer Engineering in 2018 from the Federal University of Rio Grande do Norte, Natal, Brazil. Currently, he is pursuing his M.Sc. degree in Computer Engineering at that university and he is part of the Research Group on Embedded Systems and Reconfigurable Hardware, where the main research topic is the acceleration of AI algorithms through reconfigurable computing (RC) on FPGA. His research interests include AI, embedded systems, and reconfigurable computing.

Matheus F. Torquato was born in Natal, Brazil. He received the BSc degree in Science and Technology in 2013, the B.E. degree in Computer Engineering in 2015 and the M.Sc. degree in Computer Engineering in 2017 from the Federal University of Rio Grande do Norte, Natal, Brazil. Currently, he is part of the Research Group on Embedded Systems and Reconfigurable Hardware where his main research topic is the acceleration of AI algorithms through reconfigurable computing (RC). Apart from his main research topic of AI and RC at his home university he has other research experiences such as Human-Computer Interaction at the Future Interaction Technology lab from Swansea University (Wales-UK) and Computer Vision at the Sensing and Machine Vision for Automation and Robotic Intelligence lab from the University of Ottawa (Ontario-Canada).
Marcelo A. C. Fernandes was born in Natal, Brazil. He received BS degree in Electrical Engineering in 1997, MS degree in Electrical Engineering in 1999, from the Federal University of Rio Grande do Norte, Natal, Brazil, and Ph.D. degree in Electrical Engineering in 2010, from the University of Campinas, Campinas, SP, Brazil. He is an Associate Professor in the Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte, Natal, Brazil. Currently, he is a Visiting Scholar in the John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, USA. From 2015 to 2016, he worked with a visiting researcher in Centre Telecommunication Research (CTR) at King’s College London, in London, UK. He is the leader of the Research Group on Embedded Systems and Reconfigurable Computing (RESRC) and, coordinator of the Laboratory of Machine Learning and Intelligent System (LMLIS). His research interests include AI, digital signal processing, embedded systems, reconfigurable hardware, and tactile internet. Dr. Fernandes is the author and coauthor of many scientific articles and practical studies with reconfigurable computing on FPGA to accelerate AI algorithms.

How to cite this article: da S. Medeiros DR, Torquato MF, Fernandes MAC. Embedded genetic algorithm for low-power, low-cost, and low-size-memory devices. Engineering Reports. 2020;2:e12231. https://doi.org/10.1002/eng2.12231