IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 11 September 2025, accepted 21 October 2025, date of publication 30 October 2025,
date of current version 11 November 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3626686

RESEARCH ARTICLE

FPGA-Accelerated ϵ-Greedy Ant Colony Optimisation for Maritime Bearings-Only Nonlinear Target Motion Analysis

KYRIAKOS M. DELIPARASCHOS[1], (Senior Member, IEEE), AND GABRIELE OLIVA[2], (Senior Member, IEEE)

[1]Department of Electrical Engineering and Computer Engineering and Informatics, Cyprus University of Technology, 3036 Limassol, Cyprus
[2]Department of Engineering, University Campus Bio-Medico of Rome, 00128 Rome, Italy

Corresponding author: Kyriakos M. Deliparaschos (k.deliparaschos@cut.ac.cy)

ABSTRACT Bearings-Only Target Motion Analysis (BOTMA) is challenging due to nonlinear dynamics, noisy angle-only sensing, and the nonconvexity of maximum-likelihood estimation (MLE). We present a System-on-Chip (SoC) Field-programmable gate array (FPGA) implementation of an Ant Colony Optimisation (ACO) that solves the BOTMA–MLE problem under *nonlinear* target-motion models in real time. Specifically, we consider an ϵ-greedy ACO (ϵG-ACO), where with probability $1-\epsilon$ the next component is selected according to pheromone-guided probabilities, while with probability ϵ it is chosen uniformly at random, thus balancing exploitation and exploration. The design targets a PYNQ-Z1 board (AMD Zynq-7000), where a high-level synthesis (HLS)-generated ACO intellectual property (IP) core communicates with the ARM processing system via Advanced eXtensible Interface (AXI) direct memory access (DMA) and AXI-Stream, enabling high-throughput, low-latency evaluation of candidate solutions. We validate the approach on synthetic trajectories (constant velocity, constant acceleration, and constant jerk) and on a real-world track based on Automatic Identification System (AIS) data, considering a number of parameters for the ship motion ranging from four to eight. Across all cases, the FPGA solution attains estimation accuracy comparable to Python and C++ baselines while dramatically reducing runtime: up to about $\times 119$ over a C++ implementation and up to $\times 2281$ over a Python implementation on a machine equipped with a Silicon M4 Pro 14-core CPU and 24 GB of unified RAM. On-chip power for the 8-parameter implementation is approximately 2.47 W, with the processing system dominating the dynamic power. The design fits comfortably on the XC7Z020 device, utilising about 56% of look-up tables (LUTs), 21% of block RAM (BRAM), and 85% of digital signal processors (DSPs), leaving headroom for future extensions. These results show that coupling metaheuristic optimisation with SoC-FPGA acceleration is a practical route to real-time BOTMA under realistic, nonlinear motion, particularly in resource-constrained embedded settings.

INDEX TERMS SoC-FPGA, high-level synthesis (HLS), maritime security, estimation, bearings-only target motion analysis (BOTMA), ant colony optimisation, ϵ-greedy, maximum likelihood estimation, real-time embedded systems.

I. INTRODUCTION

Bearings-only target motion analysis (BOTMA) [1], [2] plays a fundamental role in signal processing, with important applications spanning defence systems, autonomous navigation, and surveillance [3]. Interestingly, its relevance extends to underwater environments as well [4], where accurate target localisation is critical.

The core difficulty in BOTMA arises from estimating target trajectories using solely angular measurements, which are inherently nonlinear and corrupted by noise. Figure 1 provides a high-level view of the BOTMA problem, illustrating how the own-ship (i.e. the ship in charge of estimating the target's position), while following its known trajectory,

The associate editor coordinating the review of this manuscript and approving it for publication was Sotirios Goudos.

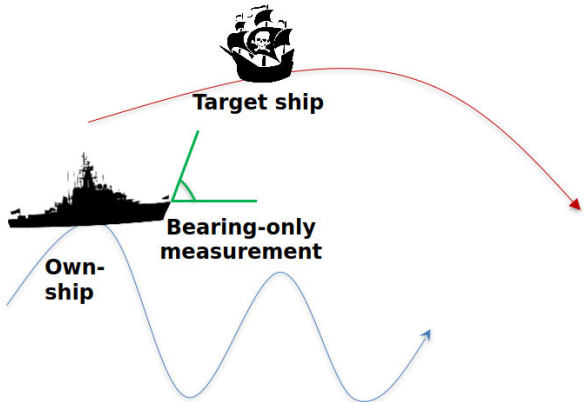


FIGURE 1. BOTMA in a nutshell: while moving, the own-ship measures only the bearing to the target; these angular data are then used to estimate the target's motion.

measures the bearings to the moving target in order to enable motion estimation.

Early studies tackled these challenges by developing closed-form expressions and iterative estimation methods such as *Maximum Likelihood Estimation* (MLE). Although these approaches provide accurate results, they often demand high computational resources and require careful initialisation to avoid convergence issues [2], [5], [6]. More recently, advancements have enabled BOTMA algorithms to operate effectively with poor-quality measurements [7], incorporate visual data [8], or manage constraints [9], [10].

Evolutionary computation methods, particularly *Ant Colony Optimisation* (ACO) [11], [12], have shown promise in addressing these computational complexities by providing robust and flexible optimisation capabilities.

In terms of hardware acceleration, *Field Programmable Gate Arrays* (FPGAs) have emerged as a powerful platform due to their parallel processing capabilities, low latency, and energy efficiency. FPGAs have been successfully applied in maritime surveillance systems [13], [14], [15], including tasks like ship detection from satellite imagery [15].

Further, FPGA-based implementations of optimisation algorithms such as Particle Swarm Optimisation (PSO) and Big Bang-Big Crunch (BB-BC) have demonstrated considerable improvements in solving nonlinear model predictive control (NMPC) problems and real-time image processing [16], [17]. Neural network approaches including Lagrange Programming Neural Networks (LPNN) and Proximal Projection Neural Networks (PPNN) have also been adapted for FPGA platforms, leveraging fixed-point arithmetic and optimised circuit designs to efficiently solve both smooth and nonsmooth optimisation tasks [17].

Regarding BOTMA specifically, [18] suggests the potential of FPGA technology in this domain, but practical implementations remain scarce. For instance, although not used for solving BOTMA, in [19] a hardware-in-the-loop simulation is presented involving also FPGAs; however, their use is limited to analogue-to-digital conversion of industrial camera signals rather than algorithmic acceleration. Despite extensive research in maritime security, existing solutions predominantly rely on conventional computational platforms (e.g. see the recent frameworks presented in [20], [21]). There is a clear gap in exploiting FPGA technology to integrate BOTMA algorithms with advanced optimisation methods like ACO. To the best of our knowledge, aside from our preliminary work [22], no comprehensive framework currently bridges the gap between theoretical BOTMA formulations using state-of-the-art optimisers and their deployment on FPGA hardware.

The novelty of this work, thus, lies in being the first FPGA-based implementation of an ϵ-greedy Ant Colony Optimisation framework tailored to the Bearings-Only Target Motion Analysis (BOTMA) problem. This contribution is distinguished by three elements: (i) the integration of a stochastic ϵ-greedy exploration strategy within a hardware-accelerated ACO loop, (ii) the adaptation of the algorithm to the nonlinear and non-convex nature of BOTMA, validated using both synthetic and real-world AIS data, and (iii) a detailed quantitative characterisation of computational speedup and energy efficiency on FPGA System-on-Chip hardware.

In this paper, we extend our preliminary work presented in [22] by introducing a comprehensive System-on-Chip (SoC) FPGA implementation of the Ant Colony Optimisation (ACO) algorithm tailored for solving the Maximum Likelihood Estimation (MLE) problem in Bearings-Only Target Motion Analysis (BOTMA). Unlike the initial design, which focused on a simplified linear motion model, our current work addresses the more complex and realistic scenario of nonlinear target motion. In particular, we adopt an ϵ-greedy ACO (ϵG-ACO) [23], [24], where pheromone-guided selection is performed with probability $1-\epsilon$ and uniform random exploration with probability ϵ, enhancing the balance between exploitation and exploration. This extension leverages the full capabilities of FPGA acceleration to optimize both performance and power efficiency, enabling real-time execution in resource-constrained environments.

The design developed in this paper is optimised for low power consumption, with the dynamic on-chip power of the SoC FPGA being 2.306 W for the 8-parameter implementation (i.e. when the target ship motion is modelled via eight parameters, as discussed later in the paper). Notably, the ACO core itself consumes only 0.769 W, after subtracting the 1.537 W consumed by the Zynq-7 PS. Very similar results are observed for the 4- and 6-parameter implementations.

By combining the flexibility of ACO with the real-time processing strengths of SoC FPGAs, our design offers an effective solution to BOTMA's computational demands. The implementation is realised on the PYNQ-Z1 platform, which integrates an AMD Zynq-7000 SoC, merging an ARM processing system with programmable logic. Simulation results validate the proposed approach, showing improvements in execution speed and energy efficiency relative to a software-only implementation on a general-purpose processor.

II. PROBLEM FORMULATION

Consider a target moving within a two-dimensional plane, whose position at discrete, uniformly spaced time intervals is described by the equations

$$\begin{cases} x_t(k) = f_x(\boldsymbol{\psi}, k, T) \\ y_t(k) = f_y(\boldsymbol{\psi}, k, T), \end{cases} \tag{1}$$

where $\boldsymbol{\psi} \in \mathbb{R}^m$ represents an unknown parameter vector, T denotes the sampling time, and k is the discrete time index characterising the sampled dynamics of the target. The functions

$$f_x : \mathbb{R}^m \times \mathbb{N} \times \mathbb{R}_{>0} \to \mathbb{R} \quad \text{and} \quad f_y : \mathbb{R}^m \times \mathbb{N} \times \mathbb{R}_{>0} \to \mathbb{R}$$

are differentiable and, possibly, nonlinear mappings that depend on the parameters, sampling interval, and index.

Within this context, we consider an own-ship platform that seeks to estimate the unknown vector $\boldsymbol{\psi}$ by collecting noisy observations sampled uniformly during its motion over the time interval $[0, k_{\max}T]$.

The nominal (noise-free) measurement model, following the example in [25], is expressed as

$$h(\boldsymbol{\psi}, k) = \operatorname{atan2}\big(y_t(k) - y_o(k), x_t(k) - x_o(k)\big), \tag{2}$$

where $x_o(k)$ and $y_o(k)$ denote the own-ship's coordinates along the x and y axes at time $t = kT$, respectively.

The actual measurements acquired by the own-ship are corrupted by noise and given by

$$z(k) = h(\boldsymbol{\psi}, k) + w(k),$$

where the noise terms $w(k)$ are independent and identically distributed Gaussian random variables with zero mean and variance σ^2, i.e. $w(k) \sim \mathcal{N}(0, \sigma^2)$.

We now address the estimation of $\boldsymbol{\psi}$ using the *Maximum Likelihood Estimation* (MLE) framework (see [26, p. 182] for more details).

The MLE estimator $\boldsymbol{\theta}^*$ is defined as the parameter value that maximizes the likelihood function of the observed data,

$$p(z_1, z_2, \ldots, z_m \mid \boldsymbol{\theta}),$$

where the maximization is performed over the admissible parameter set. For brevity, the likelihood function will be denoted simply as $p(\boldsymbol{\theta})$ whenever no ambiguity arises.

If $p(\boldsymbol{\theta})$ is differentiable, then the MLE $\boldsymbol{\theta}^*$ satisfies

$$\frac{\partial \ln p(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*} = \mathbf{0}_n, \tag{3}$$

where the solution is unique and asymptotically unbiased under standard regularity conditions.

In our problem, the likelihood function can be explicitly written as (e.g. see [2])

$$p(\boldsymbol{\theta}) = \frac{1}{(2\pi\sigma^2)^{k_{\max}/2}} \prod_{k=1}^{k_{\max}} \exp\left(-\frac{(z(k) - h(\boldsymbol{\theta}, k))^2}{2\sigma^2}\right). \tag{4}$$

Consequently, the maximization problem is equivalent to

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \ln p(\boldsymbol{\theta}).$$

By defining the cost function $\lambda(\boldsymbol{\theta}) = -\ln p(\boldsymbol{\theta})$, the estimation problem can be equivalently expressed as

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \lambda(\boldsymbol{\theta}).$$

Standard algebraic manipulations [2] yield the classical least squares formulation:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \frac{1}{2\sigma^2} \sum_{k=1}^{k_{\max}} (z(k) - h(\boldsymbol{\theta}, k))^2. \tag{5}$$

It is worth emphasizing, as noted in [27], that the MLE for bearings-only target tracking does not admit a closed-form expression; instead, a solution can be obtained via iterative numerical methods, with proper initialisation to ensure convergence.

Moreover, the cost function in (5) is generally non-convex, often necessitating the use of approximate optimisation techniques to identify a satisfactory local minimum.

Let us now conclude the section by providing an example of practical relevance that will be adopted as a benchmark in our experimental analysis later in the paper.

Example 1: Let us consider a target moving according to the following ℓ-th order polynomials

$$\begin{cases} x_t(k) = \displaystyle\sum_{h=0}^{\ell} \frac{1}{!h} \alpha_{x,h} k^h T^h \\ y_t(k) = \displaystyle\sum_{h=0}^{\ell} \frac{1}{!h} \alpha_{y,h} k^h T^h; \end{cases}$$

in this case, we have $m = 2\ell$ and

$$\boldsymbol{\psi} = \left[\alpha_{x,1}, \ldots, \alpha_{x,\ell}, \alpha_{y,1}, \ldots, \alpha_{y,\ell}\right]^\top.$$

III. FPGA-BASED SoC IMPLEMENTATION OF ANT COLONY OPTIMISATION (ACO)

This section outlines the implementation of the FPGA-based SoC accelerator for the Ant Colony optimisation (ACO) algorithm on the PYNQ-Z1 development board. The approach combines hardware acceleration with the PYNQ platform's software interface, offering efficient execution of computationally intensive tasks.

A. OVERVIEW OF THE IMPLEMENTATION

The implementation starts with simulating virtual sensor measurements on a host computer, which communicates with the PYNQ-Z1 board via Ethernet for efficient data transfer. To accelerate computation, the Ant Colony Optimisation (ACO) algorithm, known for its high repetition requirements, is offloaded to the programmable logic (PL) of the Zynq System on Chip (SoC). Within this architecture, the Zynq processing system (PS) contains the processor and DDR memory controller, while the programmable logic hosts

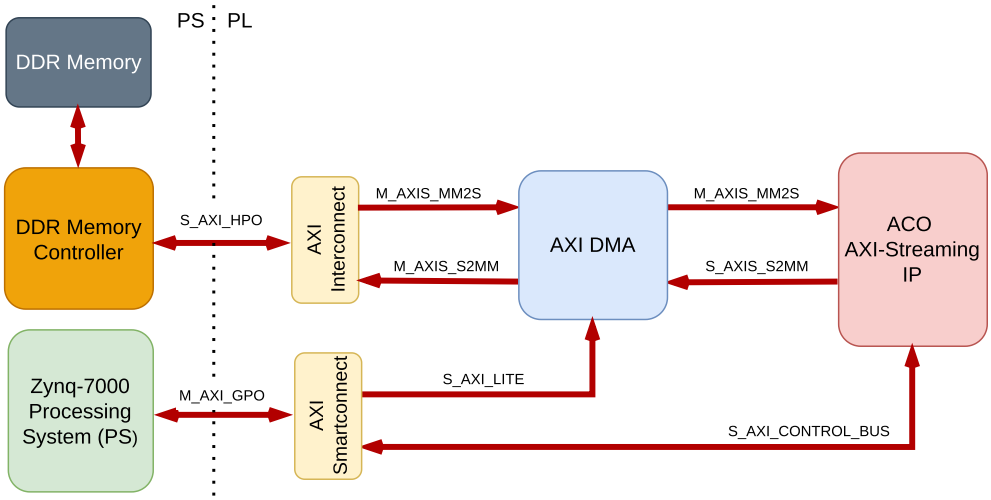


FIGURE 2. SoC FPGA-based BOTMA high-level architecture of datapath.

the AXI[1] Direct Memory Access (DMA) and AXI Data FIFO modules. Communication between the PS and PL occurs over AXI-based interfaces: AXI_Lite enables control signalling from the processor to the DMA, while AXI_MM2S and AXI_S2MM memory-mapped buses allow the DMA to read from and write to DDR memory. The AXI4-Stream interfaces, AXIS_MM2S and AXIS_S2MM, handle continuous, address-free data streams between the DMA and the FPGA accelerator. The PYNQ framework streamlines this setup by offering a Python interface to manage data movement and control tasks within the FPGA. The high-level architecture of the data-path is illustrated in Figure 2.

This integration enhances computational performance by utilising hardware acceleration, ensuring the ACO algorithm is executed efficiently. The algorithmic details are reported next.

B. ϵG-ACO ALGORITHM IMPLEMENTATION

Algorithm overview and ϵ-greedy sampling. Algorithm 1 follows the classic "construct–evaluate–update" ACO loop for continuous domains. We maintain a pheromone scalar τ_d for each decision dimension $d \in \{1, \ldots, D\}$, initialised to one. At each iteration t, a set of A ants independently construct candidate solutions

$$\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,D}) \in \mathbb{D}_1 \times \cdots \times \mathbb{D}_D,$$

where $\mathbb{D}_i = [d_{i\,\min}, d_{i,\max}]$ represents the search interval considered for the i-th dimension. Then, the objective $f(\mathbf{x}_i)$ is evaluated, and pheromones are evaporated and reinforced according to the solutions found. We track the iteration best solution $\mathbf{x}_{\text{local}}$ and update the global best solution $\mathbf{x}_{\text{best}}$ on improvement.

[1]The Advanced eXtensible Interface (AXI) is a protocol for on-chip communication buses and forms a component of the Advanced Microcontroller Bus Architecture (AMBA) specification.

ϵ-greedy construction (exploration vs. exploitation). For each ant i and dimension d, draw a uniformly random variable $r \sim \mathcal{U}(0, 1)$ and choose:

- *Exploration* (probability ϵ):

$$x_{i,d} \leftarrow d_{d,\min} + r\,(d_{d,\max} - d_{d,\min}),$$

  which ensures uniform coverage of the search interval $\mathbb{D}_d$.
- *Exploitation* (probability $1 - \epsilon$): bias the variate using the per-dimension pheromone transform

$$\hat{r} = r^{\kappa_d}, \qquad \kappa_d = \frac{1}{1 + \alpha\,\tau_d}.$$

  Based on these values, the ant selects

$$x_{i,d} \leftarrow d_{d,\min} + \hat{r}\,(d_{d,\max} - d_{d,\min}),$$

  where $\alpha > 0$ tunes pheromone influence.

Evaluation and selection. After construction, we compute $f(\mathbf{x}_i)$ (in parallel in our SoC design) and pick the iteration winner

$$\mathbf{x}_{\text{local}} \in \arg\min_{i \in \{1,\ldots,A\}} f(\mathbf{x}_i).$$

If $f(\mathbf{x}_{\text{local}}) < f(\mathbf{x}_{\text{best}})$, update $\mathbf{x}_{\text{best}}$.

Pheromone update. Pheromones undergo evaporation and deposition:

$$\tau_d \leftarrow (1 - \rho)\,\tau_d + \sum_{i=1}^{A} \Delta\tau_{i,d},$$

where

$$\Delta\tau_{i,d} = \frac{1}{1 + f(\mathbf{x}_i)}, \qquad d = 1, \ldots, D.$$

Evaporation $(1 - \rho)$ prevents unbounded growth and enables adaptation; additive reinforcement favours dimensions appearing in low-cost solutions.

Parallel realisation on FPGA. Although Algorithm 1 is written sequentially, our SoC design executes the *ant loop*

Algorithm 1 ϵ-Greedy ACO for Minimisation

Require: Objective function $f(x)$, dimensions D (number of parameters), number of ants A, maximum iterations I, evaporation rate $\rho \in (0, 1]$, exploration probability $\epsilon \in [0, 1]$, pheromone influence $\alpha > 0$, parameter search intervals $\mathbb{D}_i = [d_{i\min}, d_{i,\max}]$ for all $i \in \{1, \ldots, D\}$.

Ensure: Best solution x_{best}.

Initialise pheromone levels $\tau_d \leftarrow 1, \forall d \in \{1, \ldots, D\}$.
Set $x_{\text{best}} \leftarrow \emptyset$ and $f(x_{\text{best}}) \leftarrow \infty$.
for $t \leftarrow 1$ to I **do**
  Initialise solutions $X \leftarrow \emptyset$.
  for $i \leftarrow 1$ to A **do** ▷ Construct solutions for each ant
    for $d \leftarrow 1$ to D **do**
      choose $r_i \in [0, 1]$ uniformly at random.
      Compute biased random value

$$\hat{r}_i \leftarrow \begin{cases} r_i & \text{with probability } \epsilon \\ r_i^{\frac{1}{1+\alpha\,\tau_d}} & \text{with probability } 1-\epsilon; \end{cases}$$

      Choose $x_{i,d} \leftarrow d_{i\min} + \hat{r}_i(d_{i\max} - d_{i\min})$;
    end for
    Add solution $\boldsymbol{x}_{i,d}$ to X.
    Compute fitness $f(\boldsymbol{x}_{i,d})$.
  end for
  Find local best $\boldsymbol{x}_{\text{local}} \leftarrow \arg\min_{\boldsymbol{x}_i \in X} f(\boldsymbol{x}_i)$.
  if $f(\boldsymbol{x}_{\text{local}}) < f(\boldsymbol{x}_{\text{best}})$ **then**
    Update $\boldsymbol{x}_{\text{best}} \leftarrow \boldsymbol{x}_{\text{local}}$.
  end if
  Update pheromone levels:
  for $d \leftarrow 1$ to D **do**
    $\tau_d \leftarrow (1-\rho)\cdot\tau_d + \sum_{i=1}^{N} \frac{1}{1+f(\boldsymbol{x}_i)}$.
  end for
end for
return x_{best}

in parallel: P hardware ''ant workers'' concurrently construct $\mathbf{x}_i$, evaluate $f(\mathbf{x}_i)$ using shared read-only data (own-ship trajectory and measurements), and participate in a short reduction to update $\{\tau_d\}_{d=1}^{D}$. With full unrolling ($P = A$), the per-iteration cost drops from $\mathcal{O}(A\,N + D\,A)$ to $\mathcal{O}(N + D)$, where N is the number of measurements; a LUT-based atan2 further reduces constants while preserving accuracy.

C. DEVELOPING THE ACO IP CORE USING HIGH-LEVEL SYNTHESIS

Let us now discuss the implementation aspects of our FPGA solution. The first step in developing the FPGA accelerator is the design of the ACO IP core[2] using the AMD Vitis 2025.1 High-Level Synthesis (HLS) [28] tool. The algorithm, implemented in C++, focuses on optimising computationally intensive operations such as pheromone updates and path evaluations. These operations are synthesised into hardware logic using Vitis HLS, resulting in a reusable and highly efficient Register Transfer Level (RTL) module.

The synthesised IP core is then exported to AMD Vivado 2025.1 for integration into a complete hardware design.

D. INTEGRATING THE IP CORE INTO VIVADO DESIGN SOFTWARE

In AMD Vivado design software, the ACO IP core is integrated into a custom block design alongside the Zynq Processing System (PS). The IP core features a single high-speed input interface implemented using a 128-bit AXI-Stream for data transfer (4 inputs of 32 bits each) and one 64-bit AXI-Stream output interface (2 outputs of 32 bits each). Additionally, one AXI-Lite interface is used for control and configuration. Data movement between the Zynq PS and the ACO IP core is handled by a single AXI DMA block, which manages the 128-bit input stream and the 64-bit output stream, ensuring efficient data transfer and minimising processor overhead. The AXI interconnect remains configured to enable efficient communication between the Zynq PS and the Programmable Logic (PL), with the AXI-Lite interface directly managed by the Zynq PS for configuration and status monitoring.

The overall architecture of the accelerator integrates the Zynq PS with the ACO IP core. The Zynq PS contains both the processor and the DDR memory controller, while the programmable logic (PL) hosts the AXI DMA and AXI Data FIFO modules. The AXI DMA block provides an AXI-Stream Slave Interface (S_AXIS_S2MM) for stream-to-memory transfers and an AXI-Stream Master Interface (M_AXIS_MM2S) for memory-to-stream transfers. The 128-bit AXI-Stream input connects to the S_AXIS interface, while the 64-bit AXI-Stream output connects to the M_AXIS interface. Communication between the Zynq PS and PL occurs through various AXI interfaces, including the AXI_MM2S and AXI_S2MM memory-mapped buses, which provide DMA access to the DDR memory, and the AXIS_MM2S and AXIS_S2MM, which handle continuous, address-free data streams.

The system's control register and the scalar operand for the maximum number of entries (data size) n are managed through the S_AXI_CONTROL interface, which is part of the S_AXI_LITE interface.[3] The use of a single AXI DMA block for data handling ensures efficient throughput, with the input stream carrying three 32-bit inputs (96 bits). Since the AXI DMA requires widths in multiples of 32, the input stream is declared as 128 bits, with only the lower 96 bits utilised. The unused 32 bits, along with any associated logic resources, are optimised during synthesis to ensure the design remains efficient and compact.

Once the design is completed, an HDL wrapper is generated, followed by synthesis, implementation, and bitstream

[2] An IP core, or intellectual property core, is a reusable, pre-designed functional block of logic or data used in the development of integrated circuits (ICs) such as FPGAs and ASICs.

[3] According to AMD's guidelines, ports without a specific bundle name are grouped into the default AXI4Lite interface port, which is typically named `s_axi_control` in the RTL design.

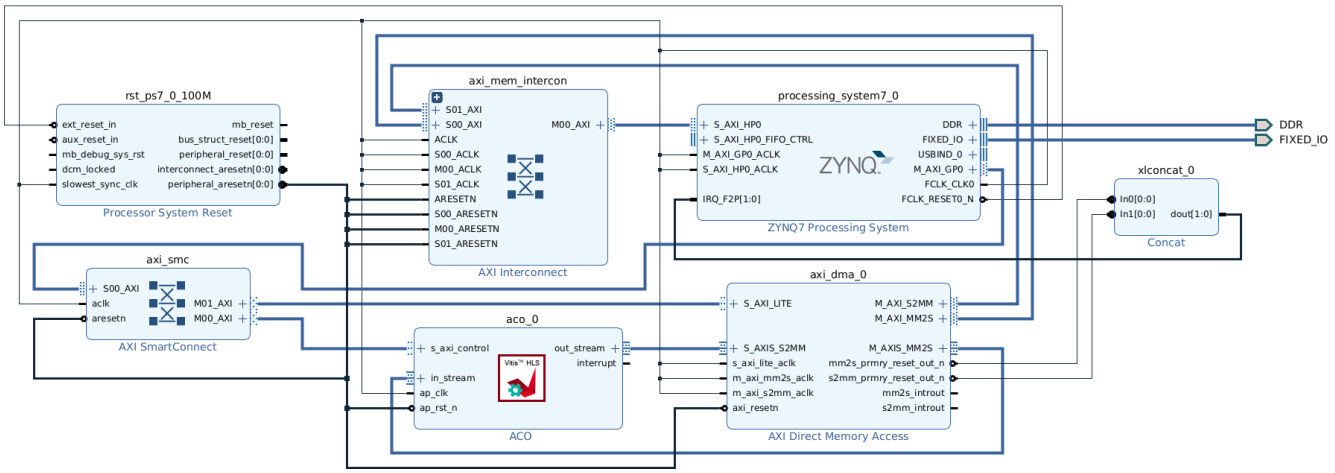


FIGURE 3. Overall system architecture and data flow.

generation. The resulting bitstream serves as a PYNQ overlay, which can be loaded onto the FPGA for execution.

The overall system architecture and data flow, implemented using AMD Vivado, is depicted in Figure 3.

E. DEPLOYING AND ACCELERATING ACO ON PYNQ

The generated PYNQ overlay is deployed on the PYNQ-Z1 and controlled using the PYNQ's Python-based interface. This interface allows the ACO algorithm to utilise the FPGA accelerator transparently, facilitating efficient execution of optimisation tasks.

The Bearings-Only Target Motion Analysis problem, which requires solving computationally demanding optimisation tasks, benefits significantly from this hardware-accelerated implementation. Parallel execution of pheromone updates and path evaluations in the PL enhances overall performance compared with a software-only solution.

IV. EXPERIMENTAL RESULTS

In this section, we analyse the performance of the proposed ACO implementation[4] with respect to three scenarios, reported in Figures 5 and 6.

The experimental setup for evaluating the proposed ACO implementation on the SoC FPGA platform uses the PYNQ-Z1 board, which features an AMD Zynq-7000 SoC integrating an ARM processor and programmable logic (see Figure 4). The FPGA board connects to a host PC via Ethernet, enabling seamless data exchange. A Jupyter Notebook environment running on the PYNQ board facilitates interactive control and monitoring of the experiment through Python scripts. The BOTMA datasets, including angular measurements and motion parameters, are transmitted from the host PC to the SoC FPGA implementation via this setup. The ACO algorithm executes on the programmable logic, and the resulting estimations are sent back to the Python environment for analysis and visualisation. This configuration allows for real-time evaluation of the ACO algorithm's performance, enabling a detailed assessment of execution speed and accuracy in a hardware-accelerated context. For comparison, we consider the results of a Python and C++ implementation of the algorithms running on a macOS machine equipped with a Silicon M4 Pro 14-core CPU and 24 GB of unified RAM.

Notice that, for all implementations, we consider 20 ants, 1000 iterations, a pheromone evaporation rate of 0.1 and $\alpha = 1$. As for the greedy search parameter ϵ, we consider the case $\epsilon = 0.2$ (i.e. 20% of uniformly random solutions).

Moreover, in the HLS (FPGA) implementation, we use a PCG-style random number generator[5] (see [29] for background on PCGs; PCG extends LCG with xorshift scrambling), which offers a favourable trade-off between statistical quality and hardware efficiency, requiring only simple arithmetic operations and a small internal state, compared with the more complex algorithms used by the `numpy.random` function in Python or the C++ `rand()` function in `cstdlib`. This hardware-oriented choice comes at the expense of slightly reduced randomness quality.

In Figure 5, we consider three distinct scenarios. In all of them, the ownship (blue curve) moves with a sinusoidal trajectory. Moreover, we assume the frame of reference is centred at the ownship's initial location and has axes that are parallel to those of Figure 5, and we simulate one hour of motion with a sampling time $T = 2s$.

As for the target ship, we assume its initial position is $x_{t0} = y_{t0} = 30000\,m$, while we consider three cases: (i) linear motion with constant velocity $\dot{x}_{t0} = 8\,m/s$, $\dot{y}_{t0} = 7\,m/s$; (ii) uniformly accelerated motion with the initial position

[4]To support reproducibility, the core project code is publicly available at https://github.com/DrFuzzy/BOTMAccel

[5]The generator employs a linear congruential core with modulus 2^{64}, $s_{n+1} = (a\,s_n + c) \bmod 2^{64}$, with multiplier $a = 6364136223846793005$ and increment $c = 1442695040888963407$ (odd). A xorshift-based permutation is then applied to the internal state, $y_n = \left(s_n \oplus (s_n \gg 18)\right)$ & `0xFFFFFF`, and scaled to $[0, 1)$ by returning $y_n/2^{24}$. This design remains lightweight in hardware (requiring one multiplication, one addition, and simple bitwise operations) while offering improved statistical quality compared with a plain LCG.

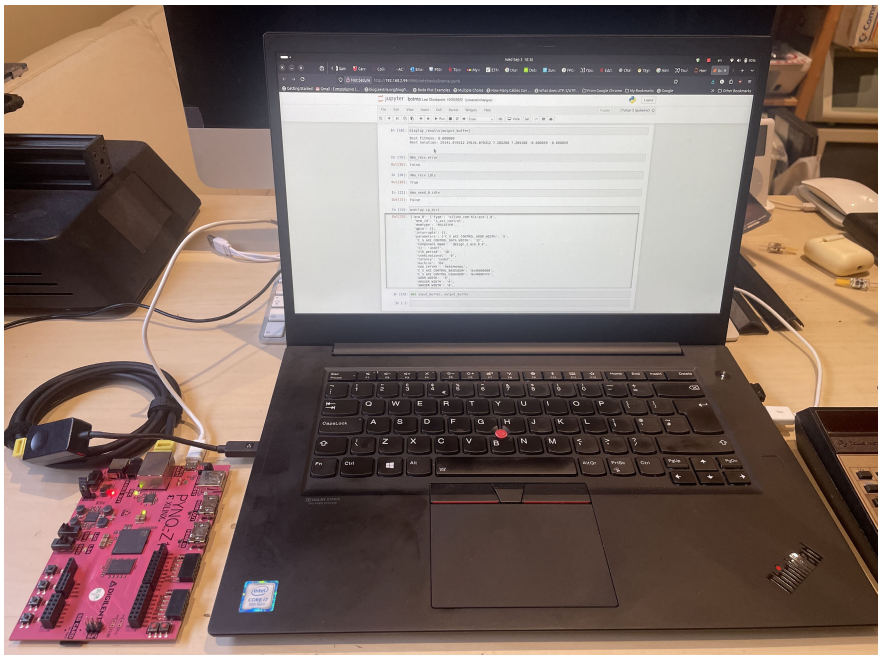


FIGURE 4. Experimental hardware setup used for all evaluations, consisting of the Pynq-Z1 development board connected to a host computer, with the overlay of the implemented design interfaced via Jupyter Notebook.

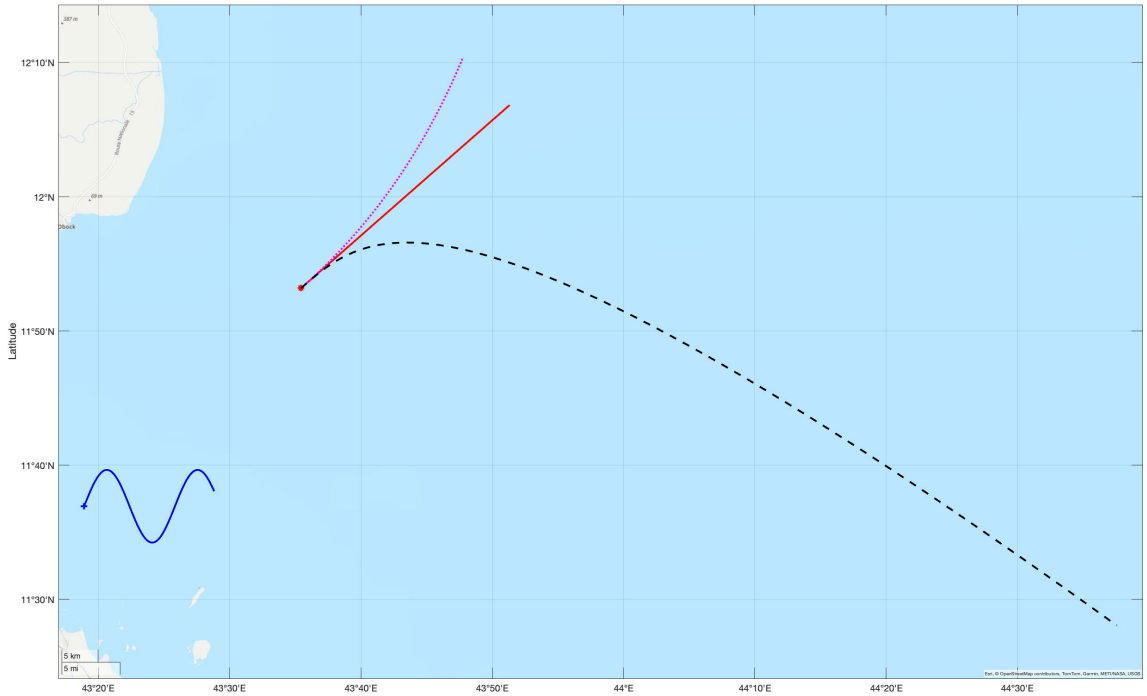


FIGURE 5. Examples of synthetic data with different motions for the target ship (duration 1 hour, sampling time $T = 2s$): linear motion (red solid line), uniformly accelerated motion (magenta dotted line), and motion with uniform jerk (black dashed line). The ownship is shown with a blue line. Latitude–longitude coordinates (degrees) are shown directly in the plots; the 5 km scale bar provides a visual distance reference.

and velocity as in the previous case and acceleration $\ddot{x}_{t0} = -10^{-3}\, m/s^2$, $\ddot{y}_{t0} = 10^{-3}\, m/s^2$; (iii) constant jerk motion with initial position and velocity as in the first case, initial acceleration $\ddot{x}_{t0} = \ddot{y}_{t0} = 10^{-3}\, m/s^2$, and constant jerk $\dddot{x}_{t0} = 10^{-5}$, $\dddot{y}_{t0} = -10^{-5}$. In all the above cases, we set the standard deviation to $\sigma = 0.01$. Moreover, for each parameter, we limit our search in a given interval, and, specifically, we consider $x_{t0}, y_{t0} \in \{20000, 40000\}\, m$, $\dot{x}_{t0}, \dot{y}_{t0} \in \{5, 10\}\, m/s$, $\ddot{x}_{t0}, \ddot{y}_{t0} \in \{-0.1, 0.01\}\, m/s^2$, and $\dddot{x}_{t0}, \dddot{y}_{t0} \in \{-0.0001, 0.0001\}\, m/s^3$.

FIGURE 6. Example featuring real trajectories (duration 1 hour, sampling time $T = 9.8467\ s$). The ownship is shown with a blue line and its start point is denoted by an asterisk. The motion of the target ship is shown in red and the initial point is shown with a circle. Latitude–longitude coordinates (degrees) are shown directly in the plots; the 50 km scale bar provides a visual distance reference.

A last example, depicted in Figure 6, features real trajectories obtained from a repository of *Automatic Identification System* (AIS) data [30]. For the sake of simplicity, we rescale them in order to obtain the same number of samples as in the previous examples. Overall, the trajectories span a time horizon of $4h\,55m\,4s$ and the uniform rescaling yields a sampling time[6] $T = 9.8467\ s$. Notice that, in this example, the trajectory of the target ship is not perfectly linear. Also in this case, for each parameter, we limit our search in a given interval that has been selected experimentally, and, specifically, we consider $x_{t0} \in \{-500000, -200000\}\,m$, $y_{t0} \in \{200000, 500000\}\,m$, $\dot{x}_{t0}, \dot{y}_{t0} \in \{0, 10\}\,m/s$, $\ddot{x}_{t0} \in \{0, 0.001\}\,m/s^2$, $\ddot{y}_{t0} \in \{-0.001, 0\}\,m/s^2$, and $\dddot{x}_{t0}$, $\dddot{y}_{t0} \in \{-0.00001, 0.00001\}\,m/s^3$.

Resource Utilisation. Tables 1–3 present the resource utilisation summary of the implemented FPGA-based SoC accelerator for Ant Colony Optimisation (ACO) in Bearings-Only Target Motion Analysis (BOTMA) with four, six, and eight parameters, respectively. The architecture integrates the Processing System (PS) and Programmable Logic (PL) components, with the ACO module serving as the computational core of the system. The implementation was carried out on the PYNQ-Z1 platform, which features an AMD Zynq-7000 SoC (part number xc7z020-clg400-1) comprising an ARM Cortex-A9 processor and FPGA fabric.

These resource utilisation results correspond to ACO implementations configured with three input data sets, each consisting of 1801 entries: *ownship_x* (the position of the ownship along the xaxis), *ownship_y* (the position of the ownship along the y-axis), and *measure* (the measured relative positions). For all three configurations (i.e. for four, six, and eight parameters), we consider two outputs: *best_solution*, which stores the estimated parameter values, and *best_fitness*, which represents the associated objective function cost.

For the 8-parameter configuration, the ACO instance utilises approximately 56% of the available slice LUTs, 27% of Flip-Flops (FFs), 21% of block RAM (BRAM) tiles, and 85% of DSP resources, highlighting its computational demands (see Table 3). For the 4- and 6-parameter configurations, the detailed resource utilisation is provided in Tables 1 and 2, respectively. The remaining peripherals, including DMA controllers, AXI interconnects, and system management cores, contribute to the overall resource utilisation with relatively lower occupancy. It is important to note that the "SoC FPGA design" row in each table represents the total resource utilisation for the entire design, including the ACO instance and all supporting components. These tables also illustrate the resource distribution between the ACO module and the supporting components within the SoC architecture.

[6]Preliminary experiments with an integer-valued sampling time T indicated marginally lower resource utilisation and power consumption. In the present work, however, we adopt a floating-point T to accommodate more general and practically relevant scenarios where the sampling interval is non-integer.

TABLE 1. Resource utilisation (implementation for four parameters).

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)	Bonded IOPADS (130)	BUFGCTRL (32)
SoC FPGA Design	**18749**	**18412**	**26**	**1**	**6451**	**18184**	**565**	**27.5**	**95**	**130**	**1**
aco_0 (design_3_aco_0_0)	15178	13474	21	1	5044	14843	335	18.0	95	0	0
axi_dma_0 (design_3_axi_dma_0_0)	1447	2176	5	0	633	1326	121	7.0	0	0	0
axi_dma_1 (design_3_axi_dma_0_1)	486	779	0	0	221	451	35	2.5	0	0	0
axi_mem_intercon (design_3_axi_mem_intercon_0)	263	360	0	0	106	241	22	0.0	0	0	0
axi_mem_intercon1 (design_3_axi_mem_intercon_1)	334	368	0	0	124	305	29	0.0	0	0	0
axi_mem_intercon2 (design_3_axi_mem_intercon1_0)	339	413	0	0	126	319	20	0.0	0	0	0
axi_smc (design_3_axi_smc_1)	677	776	0	0	283	676	1	0.0	0	0	0

TABLE 2. Resource utilisation (implementation for six parameters).

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)	Bonded IOPADS (130)	BUFGCTRL (32)
SoC FPGA Design	**24709**	**23470**	**28**	**1**	**8617**	**24005**	**704**	**28.5**	**141**	**130**	**1**
aco_0 (design_3_aco_0_0)	21160	18533	23	1	7246	20685	475	19.0	141	0	0
axi_dma_0 (design_3_axi_dma_0_0)	1449	2176	5	0	634	1328	121	7.0	0	0	0
axi_dma_1 (design_3_axi_dma_0_1)	489	779	0	0	232	454	35	2.5	0	0	0
axi_mem_intercon (design_3_axi_mem_intercon_0)	263	360	0	0	98	241	22	0.0	0	0	0
axi_mem_intercon1 (design_3_axi_mem_intercon_1)	302	367	0	0	120	274	28	0.0	0	0	0
axi_mem_intercon2 (design_3_axi_mem_intercon1_0)	343	413	0	0	144	323	20	0.0	0	0	0
axi_smc (design_3_axi_smc_1)	676	776	0	0	273	675	1	0.0	0	0	0

TABLE 3. Resource utilisation (implementation for eight parameters).

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)	Bonded IOPADS (130)	BUFGCTRL (32)
SoC FPGA design	**30524**	**28523**	**32**	**1**	**10406**	**29669**	**855**	**29.5**	**187**	**130**	**1**
aco_0 (design_3_aco_0_0)	26947	23585	27	1	9049	26322	625	20.0	187	0	0
axi_dma_0 (design_3_axi_dma_0_0)	1448	2176	5	0	663	1327	121	7.0	0	0	0
axi_dma_1 (design_3_axi_dma_0_1)	487	779	0	0	234	452	35	2.5	0	0	0
axi_mem_intercon (design_3_axi_mem_intercon_0)	263	360	0	0	105	241	22	0.0	0	0	0
axi_mem_intercon1 (design_3_axi_mem_intercon_1)	336	368	0	0	119	307	29	0.0	0	0	0
axi_mem_intercon2 (design_3_axi_mem_intercon1_0)	343	413	0	0	144	323	20	0.0	0	0	0
axi_smc (design_3_axi_smc_1)	674	776	0	0	280	673	1	0.0	0	0	0

Tables 4–6 presents the power analysis of the FPGA-based SoC implementation for the ACO algorithm applied to the BOTMA problem, considering a number of parameters ranging from four to eight. In the case of eight parameters, the total on-chip power consumption is 2.47 W, of which 2.306 W (93%) is dynamic power and 0.164 W (7%) is device static power. The dynamic power is further distributed among different components, with the highest contribution coming from the Zynq-7 Processing System (PS), consuming 1.537 W, which corresponds to about 66.6% of the dynamic power. Results for four and six parameters are comparable (see Tables 4, 5).

It is important to note that the FPGA design, which includes clocks, signals, logic, BRAM, and DSP blocks, accounts for only 33.3% (8 parameters), 18.1% (6 parameters), and 12.3% (4 parameters) of the total dynamic power. The power analysis assumes an ambient temperature of 25°C, providing a realistic estimate of power consumption under typical operating conditions. The results indicate that most of the power is consumed by the Zynq-7 PS, while the FPGA-based ACO accelerator is highly power-efficient, using only a small fraction of the total power.

Figure 7 depicts the floorplan of the implemented SoC FPGA, i.e. the physical locations and relationships of the blocks as well as the routing resources available for connecting them.

TABLE 4. Power analysis of the SoC FPGA implementation (four parameters).

Power Component	Power (W)	Percentage (%)
Total On-Chip Power: 1.92 W		
Dynamic Power	1.775	92
Clocks	0.069	4
Signals	0.064	4
Logic	0.058	3
BRAM	0.021	1
DSP	0.025	1
Zynq-7 PS	1.537	87
Device Static Power	0.146	8

TABLE 5. Power analysis of the SoC FPGA implementation (six parameters).

Power Component	Power (W)	Percentage (%)
Total On-Chip Power: 2.027 W		
Dynamic Power	1.878	93
Clocks	0.090	5
Signals	0.097	5
Logic	0.082	4
BRAM	0.017	1
DSP	0.055	3
Zynq-7 PS	1.537	82
Device Static Power	0.149	7

Performance Results. Tables 7–10 show the results for the aforementioned scenarios in terms of execution time, objective function of the MLE problem, and parameters being estimated. Specifically, the tables report the results obtained by the Python, C++, and FPGA implementations considering $m \in \{4, 6, 8\}$. The results in the tables show the

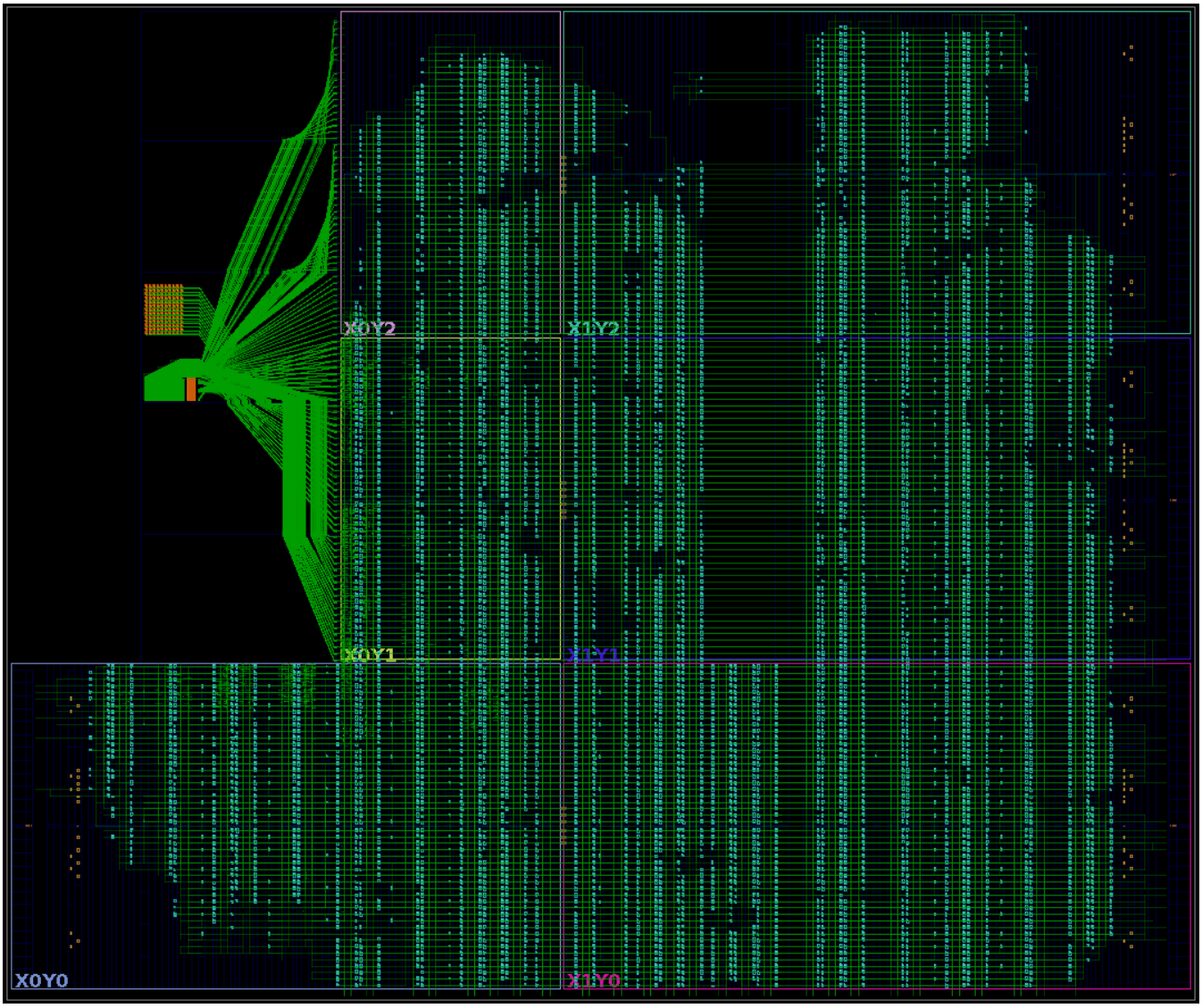


FIGURE 7. Floorplan of the implemented design (for 8 parameters) on the AMD XC7Z020CLG400-1 SoC.

TABLE 6. Power analysis of the SoC FPGA implementation (8 parameters).

Power Component	Power (W)	Percentage (%)
Total On-Chip Power: 2.47 W		
Dynamic Power	2.306	93
Clocks	0.112	5
Signals	0.299	13
Logic	0.244	11
BRAM	0.019	1
DSP	0.094	4
Zynq-7 PS	1.537	66
Device Static Power	0.164	7

average and standard deviation over $M = 10$ runs for each scenario and parameter configuration.[7]

[7]Run-to-run variability is small (see standard deviations in Tables 7–10), and observed differences across implementations are large relative to this variability; consequently, additional significance tests would not alter the conclusions and are omitted for brevity.

Interestingly, given the nonconvex nature of the problem at hand and the presence of noise, the nominal solution for the unknown parameters does not correspond to a zero objective function value, but is close to the cost of the ACO solutions.

Across implementations, the estimation accuracy (cost $\lambda(\boldsymbol{\theta})$) is largely comparable: in the constant-velocity case the FPGA yields the lowest cost for all m (0.181, 0.231, 0.510 vs. Python 0.208, 0.319, 0.650 and C++ 0.202, 0.315, 0.702), while in constant-acceleration it is best at m=6 (0.232) and within $<$ 1% of Python/C++ at m=4, with Python/C++ ahead at m=8 ($\approx$ 0.46 vs. 0.53). In the constant-jerk scenario, Python/C++ outperforms the FPGA at m=8 ($\approx$ 1.15 vs. 3.92), whereas on the real-world track with m=8 the FPGA attains the lowest cost (1.97) compared with C++ (2.57) and Python (4.71). These trends confirm that the FPGA maintains competitive accuracy across synthetic and real-world cases.

TABLE 7. Example from Fig. 5 where the target ship moves with constant velocity. Results are given in terms of mean and standard deviation over ten runs for each parameter configuration.

Implementation	m	Time $[s]$	$\lambda(\boldsymbol{\theta})$	$x_{t0}[Km]$	$y_{t0}[m]$	$\dot{x}_{t0}[m/s]$
Python	4	61.595 ± 0.6016	0.2083 ± 0.0061	31337.00 ± 1042.40	30793.39 ± 976.14	9.9798 ± 0.0124
	6	76.640 ± 0.5170	0.3478 ± 0.0271	39282.20 ± 1063.99	38437.28 ± 1549.09	7.2695 ± 1.9710
	8	88.756 ± 0.5971	0.6497 ± 0.0519	37840.22 ± 3528.68	36723.95 ± 3524.81	9.3536 ± 1.2226
C++	4	3.0286 ± 0.0108	0.2016 ± 0.0109	30586.10 ± 1685.08	30270.98 ± 1559.17	9.4738 ± 0.9732
	6	3.7224 ± 0.0046	0.3036 ± 0.0514	36572.86 ± 4940.47	36501.64 ± 4793.43	8.6285 ± 1.1577
	8	4.6215 ± 0.0081	0.7015 ± 0.0427	39060.53 ± 1505.45	38103.79 ± 2103.69	9.4781 ± 0.9144
FPGA	4	0.0389	0.1812 ± 0.0026	30647.98 ± 1365.71	30546.56 ± 1355.48	8.2625 ± 1.0826
	6	0.0390	0.2310 ± 0.0263	32698.03 ± 4640.80	32348.49 ± 3998.49	7.4897 ± 1.2161
	8	0.0392	0.5096 ± 0.1226	35203.90 ± 2626.75	34260.92 ± 2672.05	7.4149 ± 1.6219

Implementation	m	$\dot{y}_{t0}[m/s]$	$\ddot{x}_{t0}[m/s^2]$	$\ddot{y}_{t0}[m/s^2]$	$\dddot{x}_{t0}[m/s^3]$	$\dddot{y}_{t0}[m/s^3]$
Python	4	9.8229 ± 0.1990				
	6	8.1650 ± 1.7425	$3.6334e-03 \pm 5.5106e-03$	$3.8542e-03 \pm 6.4318e-03$		
	8	7.5317 ± 2.0802	$-1.6330e-03 \pm 4.8373e-03$	$4.5598e-03 \pm 6.7708e-03$	$6.4590e-05 \pm 2.6841e-05$	$7.6207e-05 \pm 3.2464e-05$
C++	4	8.9390 ± 1.3168				
	6	8.7805 ± 1.6785	$-1.3599e-03 \pm 4.4042e-03$	$-1.9797e-03 \pm 5.4812e-03$		
	8	6.6030 ± 1.6359	$1.2650e-06 \pm 5.4070e-03$	$8.4909e-03 \pm 2.3709e-03$	$6.5803e-05 \pm 3.2005e-05$	$7.6069e-05 \pm 3.5989e-05$
FPGA	4	7.4384 ± 1.2604				
	6	7.0847 ± 1.2821	$4.4250e-04 \pm 2.3840e-03$	$3.1570e-04 \pm 3.0981e-03$		
	8	7.4509 ± 1.2561	$1.9097e-03 \pm 4.7418e-03$	$2.1471e-03 \pm 5.7475e-03$	$1.9100e-05 \pm 3.7275e-05$	$2.3800e-05 \pm 4.7611e-05$

TABLE 8. Example from Fig. 5 where the target ship moves with constant acceleration. Results are given in terms of mean and standard deviation over ten runs for each parameter configuration.

Implementation	m	Time $[s]$	$\lambda(\boldsymbol{\theta})$	$x_{t0}[Km]$	$y_{t0}[m]$	$\dot{x}_{t0}[m/s]$
Python	4	61.9750 ± 0.5027	0.3311 ± 0.0102	39391.73 ± 602.92	36852.58 ± 511.09	5.6034 ± 0.2900
	6	76.5420 ± 0.7053	0.3193 ± 0.0127	38801.07 ± 872.26	37328.78 ± 987.53	8.5718 ± 1.4762
	8	89.0150 ± 0.8079	0.4620 ± 0.0185	38475.01 ± 949.27	38056.74 ± 773.79	9.4910 ± 0.5703
C++	4	3.0291 ± 0.0093	0.3294 ± 0.0078	39816.54 ± 322.50	37178.00 ± 192.86	5.5805 ± 0.2506
	6	3.7175 ± 0.0018	0.3153 ± 0.0258	36130.50 ± 5274.68	35658.54 ± 4601.26	9.4688 ± 0.8092
	8	4.6403 ± 0.0191	0.4632 ± 0.0118	39584.37 ± 427.53	39541.74 ± 356.90	9.4820 ± 1.0854
FPGA	4	0.0389	0.3316 ± 0.0104	37521.86 ± 1255.71	35286.06 ± 1189.20	5.3590 ± 0.2482
	6	0.0390	0.2316 ± 0.0243	29422.49 ± 2533.52	29429.34 ± 2136.70	8.8602 ± 0.6242
	8	0.0392	0.5271 ± 0.1655	33803.93 ± 3903.14	32503.88 ± 3976.20	7.7651 ± 1.1996

Implementation	m	$\dot{y}_{t0}[m/s]$	$\ddot{x}_{t0}[m/s^2]$	$\ddot{y}_{t0}[m/s^2]$	$\dddot{x}_{t0}[m/s^3]$	$\dddot{y}_{t0}[m/s^3]$
Python	4	9.5363 ± 0.3627				
	6	9.1067 ± 1.4582	$2.4212e-03 \pm 1.6349e-03$	$6.8687e-03 \pm 3.2864e-03$		
	8	7.1122 ± 1.3707	$7.0608e-03 \pm 3.3976e-03$	$5.4877e-03 \pm 4.4202e-03$	$5.1719e-05 \pm 4.6998e-06$	$9.0034e-05 \pm 6.7324e-06$
C++	4	9.6093 ± 0.5664				
	6	8.2517 ± 1.8680	$2.9700e-03 \pm 1.2735e-03$	$8.7344e-03 \pm 2.3647e-03$		
	8	5.9265 ± 0.7028	$9.1648e-03 \pm 1.0728e-03$	$8.8548e-03 \pm 1.6568e-03$	$5.2340e-05 \pm 3.1945e-06$	$9.2103e-05 \pm 6.0273e-06$
FPGA	4	8.6766 ± 0.5743				
	6	7.5498 ± 0.9154	$-8.4100e-05 \pm 1.9625e-03$	$2.8019e-03 \pm 3.3534e-03$		
	8	7.2650 ± 1.4728	$1.6478e-03 \pm 2.0980e-03$	$5.7629e-03 \pm 3.9230e-03$	$1.4000e-05 \pm 1.8915e-05$	$1.7600e-05 \pm 2.5505e-05$

TABLE 9. Example from Fig. 5 where the target ship moves with constant jerk. Results are given in terms of mean and standard deviation over ten runs for each parameter configuration.

Implementation	m	Time $[s]$	$\lambda(\boldsymbol{\theta})$	$x_{t0}[Km]$	$y_{t0}[m]$	$\dot{x}_{t0}[m/s]$
Python	4	61.7640 ± 0.4633	208.0274 ± 3.0609	38500.87 ± 942.39	21985.53 ± 1525.58	9.8173 ± 0.1638
	6	76.7540 ± 0.6075	1.4826 ± 0.5688	26591.08 ± 3756.87	27895.09 ± 3624.25	6.6195 ± 0.5722
	8	89.0660 ± 0.6094	1.1534 ± 0.5987	31339.62 ± 4854.72	30976.75 ± 4359.68	8.1527 ± 0.8513
C++	4	3.0303 ± 0.0148	204.3927 ± 1.1521	38932.41 ± 968.21	22186.00 ± 1028.66	9.8425 ± 0.1097
	6	3.7194 ± 0.0015	1.7638 ± 0.3877	31705.02 ± 3810.93	33698.54 ± 4301.07	8.3184 ± 1.4571
	8	4.6143 ± 0.0095	1.1563 ± 0.5991	37281.81 ± 1691.26	37427.06 ± 2424.40	8.1242 ± 1.7189
FPGA	4	0.0389	205.4878 ± 2.1941	37795.91 ± 1358.00	21175.12 ± 1265.23	9.9076 ± 0.0758
	6	0.0390	0.7850 ± 0.2722	26078.55 ± 3461.96	26575.38 ± 3682.43	6.0894 ± 1.3880
	8	0.0392	3.9171 ± 1.5045	28074.46 ± 6432.44	29062.55 ± 6267.17	7.0418 ± 1.1537

Implementation	m	$\dot{y}_{t0}[m/s]$	$\ddot{x}_{t0}[m/s^2]$	$\ddot{y}_{t0}[m/s^2]$	$\dddot{x}_{t0}[m/s^3]$	$\dddot{y}_{t0}[m/s^3]$
Python	4	5.1630 ± 0.1569				
	6	6.8592 ± 1.0750	$4.8898e-03 \pm 1.3472e-03$	$-8.9887e-03 \pm 5.6226e-04$		
	8	8.2285 ± 1.4435	$2.2826e-03 \pm 5.0854e-03$	$3.4382e-03 \pm 6.1507e-03$	$1.1377e-05 \pm 8.5107e-06$	$-1.3507e-05 \pm 4.9634e-06$
C++	4	5.0505 ± 0.0439				
	6	6.4507 ± 1.5778	$3.2339e-03 \pm 1.6679e-03$	$-9.6460e-03 \pm 4.3349e-04$		
	8	8.8125 ± 1.2030	$4.3527e-03 \pm 3.9662e-03$	$4.4167e-03 \pm 4.5944e-03$	$1.1371e-05 \pm 7.1164e-06$	$-1.5288e-05 \pm 4.2370e-06$
FPGA	4	5.0821 ± 0.0683				
	6	7.9871 ± 0.7568	$5.1190e-03 \pm 1.5173e-03$	$-9.4284e-03 \pm 4.1642e-04$		
	8	6.8795 ± 1.3398	$5.0664e-03 \pm 2.1290e-03$	$-8.9086e-03 \pm 7.3557e-04$	$-1.7000e-06 \pm 2.1777e-05$	$5.0000e-07 \pm 1.2159e-05$

Table 11 compares the average execution times of the Python and C++ implementations with the FPGA one. In spite of the low-end nature of the considered FPGA setup (the FPGA-based SoC accelerator, implemented on the PYNQ-Z1 platform, operates at a clock frequency of 100 MHz), the FPGA implementation achieves a massive improvement, ranging from 77.6× to 118.89× faster than C++ and from 1581.04× to 2281.49× faster than Python, depending on the scenario.

These remarkable performance gains highlight the advantages of hardware-based execution, specifically FPGAs, which accelerate computations by executing multiple tasks concurrently through dedicated hardware resources. The results demonstrate the effectiveness of the FPGA-based SoC solution in accelerating the ACO algorithm for the BOTMA

TABLE 10. Real World example from Fig. 6. Results are given in terms of mean and standard deviation over ten runs for each parameter configuration.

Implementation	m	Time $[s]$	$\lambda(\boldsymbol{\theta})$	$x_{t0}[Km]$	$y_{t0}[m]$	$\dot{x}_{t0}[m/s]$
Python	4	62.5410 ± 1.4695	2.3153 ± 1.2695	-421939.63 ± 12220.58	4831175.59 ± 28902.72	8.3219 ± 1.4779
	6	77.0410 ± 0.3500	1.2836 ± 0.6696	-413622.13 ± 25677.02	4843457.06 ± 59414.85	8.5081 ± 1.1232
	8	88.8720 ± 0.7201	4.7122 ± 2.1099	-368157.99 ± 23549.71	4956352.57 ± 38186.37	5.6648 ± 2.1687
C++	4	3.0233 ± 0.0153	1.4584 ± 0.8344	-417104.70 ± 14455.05	4844483.00 ± 30827.42	9.0386 ± 0.6978
	6	3.7378 ± 0.0150	0.3432 ± 0.0639	-431339.10 ± 7266.17	4800381.00 ± 18683.36	8.3927 ± 1.0397
	8	4.6412 ± 0.0340	2.5710 ± 0.9462	-360196.50 ± 17869.57	4953084.00 ± 39868.03	6.0868 ± 3.4071
FPGA	4	0.0389	3.1145 ± 1.9120	-420576.11 ± 16036.51	4824036.45 ± 27622.95	7.5286 ± 1.2559
	6	0.0390	2.0041 ± 0.8075	-392306.15 ± 32307.08	4878392.45 ± 67762.29	8.6255 ± 1.1597
	8	0.0392	1.9662 ± 0.8887	-397244.85 ± 17425.66	4874992.20 ± 36638.49	8.8421 ± 0.8157

Implementation	m	$\dot{y}_{t0}[m/s]$	$\ddot{x}_{t0}[m/s^2]$	$\ddot{y}_{t0}[m/s^2]$	$\dddot{x}_{t0}[m/s^3]$	$\dddot{y}_{t0}[m/s^3]$
Python	4	1.6121 ± 1.0989				
	6	5.7633 ± 2.5664	$5.1867e-04 \pm 2.4560e-04$	$-2.0499e-04 \pm 1.7253e-04$		
	8	5.5680 ± 3.2075	$3.3465e-04 \pm 2.3150e-04$	$-5.6538e-04 \pm 3.5273e-04$	$3.6168e-06 \pm 1.7127e-06$	$2.9060e-06 \pm 1.3628e-06$
C++	4	1.6938 ± 1.5710				
	6	8.7484 ± 0.9935	$8.2386e-04 \pm 1.2998e-04$	$-9.6660e-05 \pm 5.3701e-05$		
	8	7.0404 ± 2.3463	$5.7956e-04 \pm 2.4975e-04$	$-4.7105e-04 \pm 2.5433e-04$	$2.4062e-06 \pm 1.6632e-06$	$1.8100e-06 \pm 1.2986e-06$
FPGA	4	1.1607 ± 0.8199				
	6	5.0063 ± 2.3744	$6.0360e-04 \pm 2.3670e-04$	$-3.3780e-04 \pm 1.8957e-04$		
	8	4.9936 ± 3.1682	$5.3400e-04 \pm 3.1893e-04$	$-3.6030e-04 \pm 2.3135e-04$	$3.5000e-06 \pm 4.0346e-06$	$-1.6000e-06 \pm 6.0037e-06$

TABLE 11. ACO average speedup comparison with respect to Python and C++.

	Const. vel.			Const. acc.			Const. jerk			Real World		
Implementation	m=4	m=6	m=8	m=4	m=6	m=8	m=4	m=6	m=8	m=4	m=6	m=8
Python	1581.04	1963.19	2273.55	1590.8	1960.68	2280.19	1585.38	1966.11	2281.49	1605.33	1973.46	2276.52
C++	77.74	95.35	118.38	77.75	95.23	118.86	77.78	95.28	118.2	77.6	95.75	118.89

problem, making it well-suited for real-time or near-real-time applications.

Fixed-Point Arithmetic and Accuracy. To optimise computational efficiency on the FPGA, all arithmetic operations within the ACO IP core were implemented using fixed-point rather than floating-point representation. This approach reduces logic and DSP utilisation, data transfer volume, and latency, while maintaining sufficient numerical precision for stable optimisation performance. Fixed-point arithmetic also improves data throughput across the 128-bit AXI-Stream interface by reducing the number of effective fractional bits processed and stored, thereby enabling more efficient use of on-chip memory and DMA bandwidth. Despite this simplification, the FPGA implementation achieves comparable or even superior estimation accuracy. For instance, in all the synthetic examples the FPGA attains, on average (over all scenarios and parameter choices), a relative accuracy[8] improvement for the target's initial position ($x_0^{\mathrm{gt}} = 30000\,m$) of about 28.25% with respect to Python and 24.08% with respect to C++ (both using floating point arithmetic). The variability across experiments is non-negligible, with standard deviations of about ±35.11% (Python) and ±26% (C++).

V. CONCLUSION

In this work, we presented a novel FPGA-based SoC accelerator for ACO applied to the BOTMA problem. Our implementation, deployed on the PYNQ-Z1 platform, demonstrates the effectiveness of integrating evolutionary optimisation algorithms with hardware accelerators for real-time signal processing tasks. Through simulations, we validated the proposed solution's ability to efficiently solve the Maximum Likelihood Estimation (MLE) problem in BOTMA, achieving significant performance improvements over traditional software-based implementations.

The experimental results highlight the benefits of our FPGA-accelerated ACO approach. Specifically, the FPGA implementation, despite the low-end nature of the FPGA board used for our experimental analysis, achieved up to a 2281.49× speedup over a Python-based solution and a 118.89× speedup compared with a C++ software model, all while maintaining low power consumption and efficient resource utilisation. The power analysis showed that the FPGA design accounted for only 33.3% (8 parameters), 18.1% (6 parameters), and 12,3% (4 parameters) of the total dynamic power, underscoring the efficiency of hardware-based parallelism in executing computationally intensive optimisation tasks.

In summary, beyond the demonstrated performance and efficiency gains, this work represents the first FPGA-based implementation of an ϵ-greedy Ant Colony Optimisation framework for BOTMA. Its novelty lies in embedding a stochastic exploration–exploitation strategy within a hardware-accelerated loop, adapting it to nonlinear BOTMA, and validating it on both synthetic and real AIS trajectories.

Although the proposed design achieves real-time performance and competitive accuracy, its current scope is limited to single-target tracking and assumes idealised bearing measurements. Further validation on multi-target scenarios and more complex noise environments remains an open direction.

Future work will explore additional enhancements, such as integrating multi-objective optimisation techniques, refining pheromone update strategies for faster convergence, and

[8] We compute the accuracy metric as $\sqrt{|\bar{x}_0 - x_0^{\mathrm{gt}}|^2 + \sigma^2}$, where $\bar{x}_0$ and σ are the mean and standard deviation of the estimated initial position along the x-axis, and x_0^{gt} is the ground truth. This corresponds to the root mean square error (RMSE).

extending the approach to multi-target tracking scenarios. We also plan to conduct systematic comparisons with other FPGA-accelerated metaheuristics (e.g. PSO, GA) to better situate the proposed design within the broader landscape of hardware-accelerated optimisation. Additionally, we plan to investigate adaptive hardware configurations using partial reconfiguration techniques to further optimise resource utilisation and power efficiency.

REFERENCES

[1] S. Nardone, A. Lindgren, and K. Gong, "Fundamental properties and performance of conventional bearings-only target motion analysis," *IEEE Trans. Autom. Control*, vol. AC-29, no. 9, pp. 775–787, Sep. 1984.

[2] A. Farina, "Target tracking with bearings—Only measurements," *Signal Process.*, vol. 78, no. 1, pp. 61–78, Oct. 1999.

[3] M. Ebrahimi, M. Ardeshiri, and S. A. Khanghah, "Bearing-only 2D maneuvering target tracking using smart interacting multiple model filter," *Digit. Signal Process.*, vol. 126, Jun. 2022, Art. no. 103497.

[4] D. Sun, Y. Zhang, T. Teng, and L. Gao, "Underwater Doppler-bearing maneuvering target motion analysis based on joint estimated adaptive unscented Kalman filter," *J. Acoust. Soc. Amer.*, vol. 154, no. 5, pp. 2843–2857, Nov. 2023.

[5] S. C. Nardone and M. L. Graham, "A closed-form solution to bearings-only target motion analysis," *IEEE J. Ocean. Eng.*, vol. 22, no. 1, pp. 168–178, Jan. 1997.

[6] T. L. Song and T. Y. Um, "Practical guidance for homing missiles with bearings-only measurements," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 32, no. 1, pp. 434–443, Jan. 1996.

[7] Y. Zhang, C. Wang, Q. Zhang, L. Da, and Z. Jiang, "Bearing-only motion analysis of target based on low-quality bearing-time recordings map," *IET Radar, Sonar Navigat.*, vol. 18, no. 5, pp. 765–781, May 2024.

[8] Z. Ning, Y. Zhang, J. Li, Z. Chen, and S. Zhao, "A bearing-angle approach for unknown target motion analysis based on visual measurements," *Int. J. Robot. Res.*, vol. 43, no. 8, pp. 1228–1249, Jul. 2024.

[9] G. Oliva, A. Farina, and R. Setola, "Intelligence-aware batch processing for TMA with bearings-only measurements," *Sensors*, vol. 21, no. 21, p. 7211, Oct. 2021.

[10] X. Wu and X. Wu, "Bearings-only target motion analysis with nonlinear inequality constraints using two arrays," *Proc. SPIE*, vol. 12983, pp. 637–643, Jan. 2024.

[11] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Cambridge, MA, USA: MIT Press, 2004.

[12] M. Dorigo, M. Birattari, and T. Stützle, "Ant colony optimization," *IEEE Comput. Intell. Mag.*, vol. 1, no. 4, pp. 28–39, Apr. 2006.

[13] A. Lestari, D. D. Patriadi, I. H. Putri, B. Harnawan, O. D. Winarko, W. Sediono, and M. A. K. Titasari, "FPGA-based SDR implementation for FMCW maritime surveillance radar," in *Proc. Int. Conf. Radar, Antenna, Microw., Electron., Telecommun. (ICRAMET)*, Oct. 2017, pp. 15–20.

[14] P. Uran, "Design of an FPGA-based data acquisition system for a shore-based maritime radar network," M.S. thesis, Fac. Inf. Technol. Elect. Eng., Dept. Electron. Syst., Norwegian Univ. Sci. Technol., Trondheim, Norway, 2021.

[15] X. Huang, K. Xu, J. Chen, A. Wang, S. Chen, and H. Li, "Real-time processing of ship detection with SAR image based on FPGA," in *Proc. IGARSS-IEEE Int. Geosci. Remote Sens. Symp.*, Jul. 2024, pp. 8954–8957.

[16] M. Psarakis, A. Dounis, A. Almabrok, S. Stavrinidis, and G. Gkekas, "An FPGA-based accelerated optimization algorithm for real-time applications," *J. Signal Process. Syst.*, vol. 92, no. 10, pp. 1155–1176, Oct. 2020.

[17] R. Xiao, X. He, T. Huang, and J. Yu, "FPGA implementation of classical dynamic neural networks for smooth and nonsmooth optimization problems," *IEEE Trans. Sustain. Comput.*, vol. 9, no. 2, pp. 197–210, Mar. 2024.

[18] M. A. Nuhoglu and H. A. Cirpan, "Source localization using changing rate of phase difference only: A convex optimization approach," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 61, no. 2, pp. 3840–3851, Apr. 2025.

[19] J. Wang, D. Zhou, X. Zou, R. Du, and J. Liu, "Multi-target passive tracking for multi-spacecraft with limited fields-of-view," *Aerosp. Sci. Technol.*, vol. 166, Nov. 2025, Art. no. 110546.

[20] M. Chen, J. Sun, K. Aida, and A. Takefusa, "Weather-aware object detection method for maritime surveillance systems," *Future Gener. Comput. Syst.*, vol. 151, pp. 111–123, Feb. 2024.

[21] S. Li, X. Cao, and Z. Zhou, "Research on inshore ship detection under nighttime low-visibility environment for maritime surveillance," *Comput. Electr. Eng.*, vol. 118, Aug. 2024, Art. no. 109310.

[22] K. M. Deliparaschos, R. Setola, and G. Oliva, "An FPGA-based SoC accelerator for ant colony optimisation in bearings-only target motion analysis," in *Proc. 33rd Medit. Conf. Control Autom. (MED)*, Jun. 2025, pp. 132–137.

[23] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 53–66, Apr. 1997.

[24] Y. Liu, B. Cao, and H. Li, "Improving ant colony optimization algorithm with epsilon greedy and levy flight," *Complex Intell. Syst.*, vol. 7, no. 4, pp. 1711–1722, Aug. 2021.

[25] M. Mallick, "A note on bearing measurement model," *Mach. Eng.*, vol. 10, pp. 1–2, Feb. 2018.

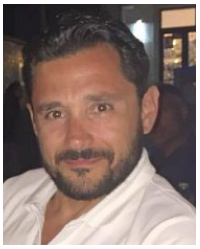
[26] S. M. Kay, *Statistical Signal Processing: Estimation Theory*, vol. 1. Upper Saddle River, NJ, USA: Prentice-Hall, 1993.

[27] K. Doğançay, "On the efficiency of a bearings-only instrumental variable estimator for target motion analysis," *Signal Process.*, vol. 85, no. 3, pp. 481–490, Mar. 2005.

[28] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[29] M. E. O'Neill, "PCG: A family of simple fast space-efficient statistically good algorithms for random number generation," Harvey Mudd College, Claremont, CA, USA, Tech. Rep. HMC-CS-2014-0905, Sep. 2014.

[30] P. Gloaguen, L. Chapel, C. Friguet, and R. Tavenard, "Scalable clustering of segmented trajectories within a continuous time framework: Application to maritime traffic data," *Mach. Learn.*, vol. 112, no. 6, pp. 1975–2001, Jun. 2023.



KYRIAKOS M. DELIPARASCHOS (Senior Member, IEEE) received the B.Eng. degree (Hons.) in electronics engineering from De Montfort University, U.K., the M.Sc. degree in mechatronics the National Technical University of Athens (NTUA), Greece, and the Ph.D. degree from the Department of Signals, Control, and Robotics, School of Electrical and Computer Engineering, NTUA. He is currently a Special Teaching Staff Member with the Department of Electrical and Computer Engineering and Informatics, Cyprus University of Technology (CUT). He has held academic and research positions at New York College and IST College, Athens, NTUA, CUT, Trinity College Dublin, and Cranfield University. His research interests include embedded systems, reconfigurable hardware acceleration, sensor fusion, resilient cyber-physical systems, intelligent control, real-time robotic navigation, and medical robotics for minimally invasive surgery.

GABRIELE OLIVA (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science and automation engineering from the University Roma Tre of Rome, Italy, in 2008 and 2012, respectively. He is currently an Associate Professor of automatic control with the University Campus Bio-Medico of Rome, Italy, where he directs the Complex Systems and Security Laboratory (CoserityLab). His main research interests include distributed multi-agent systems, optimisation, decision-making, maritime situation awareness, and critical infrastructure protection. Since 2022, he has been an Associate Editor for IEEE Control Systems Letters journal. Moreover, he has been an Associate Editor for the Conference Editorial Board of the IEEE Control Systems Society and *PLOS One*.