BABEŞ BOLYAI UNIVERSITY, CLUJ NAPOCA, ROMÂNIA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

# TITLE

– ITSG report –

**Team members**
David Szilagyi, SIA, 251, david.szilagyi2@stud.ubbcluj.ro
Razvan Filea, SIA, 251, razvan.filea@stud.ubbcluj.ro
Armin Torok, SIA, 251, armin.torok@stud.ubbcluj.ro

2025-2026

**Abstract**

Intelligent perception is a core requirement for autonomous driving, where multiple vision tasks—lane detection, object tracking, and traffic sign recognition—must operate reliably in real time. This project presents an integrated **machine learning–based perception system** built upon a hybrid **Python–Rust architecture**, combining the flexibility of Python's ML ecosystem with the efficiency and determinism of Rust. The proposed system serves as both a research and deployment framework for developing and evaluating real-time vision algorithms on autonomous vehicles.

**Relevance and importance.** Traditional Python pipelines simplify prototyping but struggle with concurrency and latency when several neural networks run in parallel. By embedding these ML modules into a Rust-backed shared-memory infrastructure using a **partial-blocking inter-process communication (IPC)** strategy, the system achieves high throughput without sacrificing development simplicity. It enables the use of advanced deep learning models in a structure suitable for embedded and safety-critical robotics applications.

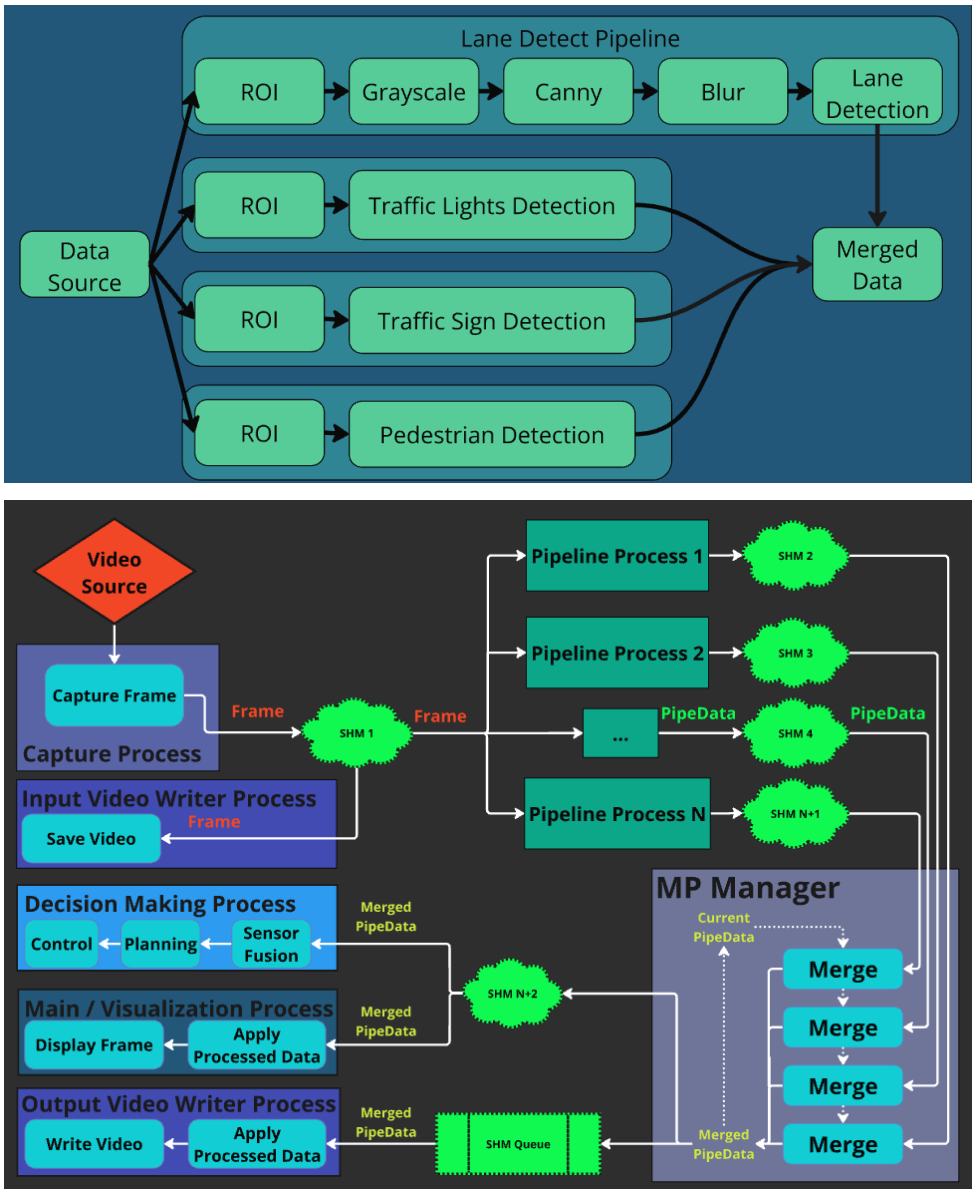
**Intelligent methods used.** The perception stack employs **YOLOv8** models, fine-tuned on *hand-crafted datasets* derived from videos recorded in manual (RC) mode on the autonomous vehicle platform. The collected footage was passed through a custom augmentation and labeling pipeline to increase dataset diversity and accuracy. Each trained model specializes in a different perception task (lane detection, object recognition, sign classification), and all are executed concurrently inside the shared Python–Rust runtime, coordinated by the Partial-Blocking communication policy.

**Data used.** Training data originates from real-world miniature driving experiments on a 1:10 scale platform. The runtime system was tested on Jetson AGX Orin and desktop CUDA/non-CUDA workstations.

**Results.** The hybrid framework delivers real-time inference performance between 35–90 FPS while reducing inter-process latency by up to 75% compared to standard multiprocessing. This allowed the vehicle to detect all relevant features with full accuracy (although not confidence) at 35 FPS, during

closed-loop autonomous driving tests. Beyond speed, it provides a seamless interface for ML development—models can be retrained or swapped in Python without modifying the Rust layer—allowing fast experimentation and reliable deployment within a unified environment.

**Graphical Abstract**




*YOLOv8-based perception modules (object, lane, sign detection) integrated in a hybrid Python–Rust real-time pipeline with shared-memory communication.*

# Contents


**1 Introduction** **1**
- 1.1 What? Why? How? . . . 1
- 1.2 Paper structure and original contribution(s) . . . 2

**2 Scientific Problem** **3**
- 2.1 Problem Definition . . . 3

**3 State of the art/Related work** **5**

**4 Investigated approach** **7**
- 4.1 Overview . . . 7
- 4.2 ML models and data pipeline . . . 8
- 4.3 Producer–Consumer Model . . . 8
  - 4.3.1 Partial-Blocking Algorithm . . . 9
- 4.4 Worked Example (Trace) . . . 10

**5 Application (numerical validation)** **11**
- 5.1 Methodology . . . 11
- 5.2 Data . . . 11
- 5.3 Results . . . 11
- 5.4 Discussion . . . 12

**6 SWOT Analysis** **13**

**7 Conclusion and future work** **14**

**8 Latex examples** **15**

# List of Tables

8.1 The parameters of the PSO algorithm (the micro level algorithm) used to compute the fitness of a GA chromosome. . . . . 16

# List of Figures

4.1 Memory layout of `SharedMessage`: fields are aligned to enable atomic access to *Message Version*, while the *Mutex* and condition variables reside in shared memory. . . . . 7

8.1 The evolution of the swarm size during the GA generations. This results were obtained for the $f_2$ test function with 5 dimensions. . . . . 15

# List of Algorithms

1 Partial-Blocking Producer–Consumer Perception Runtime . . . . . 9
2 SGA - Spin based Genetic AQlgorithm . . . . . 16

# Chapter 1

# Introduction

## 1.1 What? Why? How?

**What is the problem?** Autonomous driving requires real-time understanding of the vehicle's surroundings: detecting lanes, recognizing signs, tracking dynamic obstacles, and fusing multiple sensor inputs. These perception tasks are typically solved using deep learning methods—convolutional neural networks (CNNs) or transformer-based vision models—that must run efficiently and predictably. However, executing several ML models concurrently introduces synchronization delays and communication overhead, particularly in Python-based prototypes. The scientific problem addressed here is how to combine the flexibility of ML model development with the deterministic performance required for real-time robotic operation.

**Why is it important?** The quality of perception directly determines driving safety. An accurate but slow object detector can be less useful than a slightly less precise one that operates in real time. Researchers often face a trade-off between rapid experimentation (using Python ML frameworks such as PyTorch or TensorFlow) and deployment efficiency (which requires low-level optimization). A unified framework that keeps Python for development but offloads concurrency and memory management to Rust enables reproducible ML experiments that scale smoothly to embedded platforms.

**How do we address it?** Our solution integrates multiple trained ML models into a **hybrid Python–Rust architecture** that manages all data flow and synchronization through shared memory. The perception modules—each responsible for a distinct vision task—run as independent processes connected by a *Partial-Blocking Producer–Consumer* communication model. This mechanism allows fast inference branches (e.g., lane detection) to update continuously while slower ones (e.g., sign recognition) operate asynchronously, without stalling the system. Rust provides atomic synchronization primitives and watchdog-based runtime verification, while Python handles model inference, visualiza-

tion, and logging. Together, they deliver both development agility and real-time performance.

**Main findings.** The proposed architecture sustains real-time inference across heterogeneous ML modules with up to fourfold higher throughput compared to standard multiprocessing. It simplifies experimentation—ML models can be replaced or retrained in Python without changing the low-level code—while maintaining deterministic timing during deployment. The project demonstrates that intelligent perception does not rely solely on model accuracy: the surrounding architecture can itself embody "intelligence," enabling ML-driven systems to operate safely, efficiently, and predictably in real time.

## 1.2 Paper structure and original contribution(s)

The research presented in this paper advances the theory, design, and implementation of several particular models.

The main contribution of this report is to present an intelligent algorithm for solving the problem of ....

The second contribution of this report consists of building an intuitive, easy-to-use and user friendly software application. Our aim is to build an algorithm that will help ....

The third contribution of this thesis consists of ....

The present work contains *xyz* bibliographical references and is structured in five chapters as follows.

The first chapter/section is a short introduction in ....

The second chapter/section describes ....

The chapter/section 4 details ....

# Chapter 2

# Scientific Problem

## 2.1 Problem Definition

**Description.** The core problem addressed in this project is the design of an **intelligent perception system** capable of understanding the driving environment of an autonomous vehicle in real time. This requires detecting relevant visual cues—such as lanes, traffic signs, and obstacles—from a continuous video stream, and updating this understanding quickly enough to guide safe vehicle behavior. The challenge lies in running multiple perception models simultaneously while maintaining both accuracy and responsiveness under varying computational loads.

**Why an intelligent algorithm is required.** Such a system cannot rely only on static, rule-based logic: environmental variability, lighting changes, and partial occlusions demand **adaptive, data-driven models**. Machine learning, particularly deep learning with convolutional neural networks (CNNs), provides the capacity to learn visual features directly from data rather than relying on manually engineered ones. By learning generalizable patterns, the models can recognize objects and lane markings across conditions that are too diverse or complex to handle explicitly through traditional algorithms.

**Methods and trade-offs.** In this project, we employ fine-tuned **YOLOv8** models trained on handcrafted datasets generated from real driving footage recorded on a 1:10 scale autonomous platform in remote-controlled mode. Each video sequence was passed through a custom augmentation and labeling pipeline, creating a dataset representative of real-world conditions (e.g., shadows, reflections, variable camera angles). These models are integrated into a shared runtime, where object detection, lane segmentation, and sign recognition operate in parallel. This configuration allows fast models to update more frequently while slower ones contribute asynchronously, maintaining global perception consistency.

Classical computer vision methods (e.g., thresholding, Hough transforms, contour-based detection) could, in principle, address individual perception tasks. However, they lack the robustness to lighting variations, scale changes, and occlusions that occur in real-world conditions. In contrast, deep learning models adapt through training and can continuously improve as new data is collected—making them inherently more suitable for long-term autonomous operation.

**Significance.** This problem is both practically and scientifically relevant. Practically, it underpins the safe navigation of autonomous vehicles in dynamic environments. Scientifically, it explores the intersection between *learning-based perception* and *system-level optimization*, aiming to prove that machine learning models can coexist with strict real-time constraints when supported by a suitable hybrid architecture. Solving this problem provides a foundation for generalizing ML-based perception to other robotics domains that demand both high accuracy and performance.

# Chapter 3

# State of the art/Related work

Research on autonomous vehicle perception has evolved along two major directions: (1) the improvement of learning models for visual understanding, and (2) the design of runtime architectures that enable those models to operate under real-time constraints. However, most studies emphasize one aspect while treating the other as secondary, resulting in high-accuracy models that are difficult to deploy efficiently, or optimized systems that rely on simplified inference pipelines. Our work seeks to bridge this gap by integrating full-precision deep learning models into an execution framework explicitly optimized for low-latency, concurrent processing.

On the perception side, Rosique et al. [?] survey sensor modalities and vision architectures for autonomous driving, while Gu et al. [?] and Shahian Jahromi et al. [?] propose multi-sensor fusion pipelines designed for responsiveness and robustness. Ros et al. [?] highlight hybrid offline–online perception paradigms that improve reliability across varying conditions. These contributions advance detection and fusion accuracy but typically assume generic middleware layers that impose non-negligible communication costs.

System-oriented research, by contrast, addresses those architectural limitations. Early embedded vision works such as Sridharan and Stone [?] examined the trade-off between latency and throughput on constrained platforms, and frameworks like ROS 2 [?] and RViz [?] exemplify distributed modularity and built-in visualization. Yet, these solutions often rely on serialized message passing that limits scalability for high-frame-rate video and multiple concurrent models.

Recent efforts in multi-language and hybrid runtimes aim to merge high-level ML workflows with low-level efficiency. PyO3 [?] demonstrates a robust interface between Python and Rust, enabling flexible development with near-native performance. Scott and Brown [?] establish theoretical foundations for shared-memory synchronization, which inform modern approaches to concurrent data exchange. Building on these principles, our architecture embeds YOLOv8-based perception models directly within

a partial-blocking, shared-memory environment, thereby combining the flexibility of ML research code with the determinism of optimized real-time systems.

# Chapter 4

# Investigated approach

## 4.1 Overview

We implement a modular perception stack where multiple *YOLO* models (traffic sign, light and pedestrian detection) and a lane–geometry detection module run in parallel as independent branches. Python hosts the ML code (model loading, pre/post-processing), while a Rust-backed shared–memory layer, called *SharedMessage*[!!!!Add reference to paper here] coordinates data exchange and synchronization.

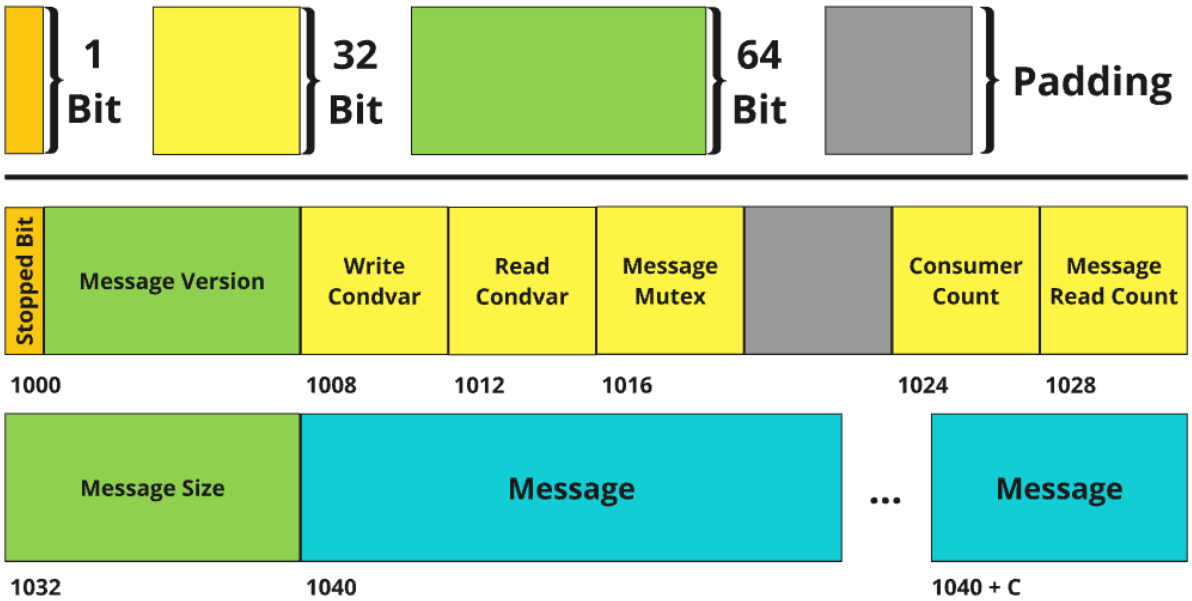



Figure 4.1: Memory layout of `SharedMessage`: fields are aligned to enable atomic access to *Message Version*, while the *Mutex* and condition variables reside in shared memory.

## 4.2 ML models and data pipeline

- **Models.** YOLOv8 variants, fine-tuned for traffic sign, traffic light, pedestrian/object detection. Lane detection uses a custom geometric algorithm based on edge detection and Hough transforms, which in our testing proved faster and more reliable than learning-based lane segmentation models.
- **Data.** Handcrafted datasets from RC-mode onboard videos, processed by an augmentation & labeling pipeline to increase diversity and accuracy.
- **Inference I/O.** Each branch consumes the latest RGB frame $f_t$; produces task-specific predictions (boxes/classes for objects/signs; line/curve params for lanes) plus timestamps.

## 4.3 Producer–Consumer Model

A key component of the perception runtime is the **Producer–Consumer model**, which governs the data flow between the frame capture process (producer) and the perception branches (consumers). Depending on timing and synchronization requirements, the system supports three strategies:

- **Non-blocking:** The producer continuously publishes new frames without waiting for any consumer. Consumers process the most recent frame independently. This mode offers maximum responsiveness but may lead to desynchronized outputs—suitable for live monitoring or non-critical visualization.
- **Full-blocking:** The producer waits until all consumers have finished processing the current frame before publishing the next. This guarantees frame-level consistency across all modules but reduces throughput. It is mainly used in testing, debugging, or analysis scenarios.
- **Partial-blocking (proposed):** The producer may advance once at least one consumer has *started* processing the current frame. This minimizes idle time while ensuring every frame is processed by at least one consumer. It achieves a balance between responsiveness and consistency, ideal for real-time autonomous driving, where some ML branches (e.g., lane tracking) must update faster than others (e.g., sign recognition).

### 4.3.1 Partial-Blocking Algorithm

**Algorithm 1** Partial-Blocking Producer–Consumer Perception Runtime

**Variables / Notation**
$F = \{f_t\}$: camera frame stream; $n$: number of consumer branches
$C = \{c_1, \ldots, c_n\}$: consumers (e.g., *objects*, *signs*, *lanes*)
$v \in \mathbb{Z}_{\geq 0}$: atomic version of the current raw frame in shared memory
$f$: frame written by Producer for version $v$
$\mathcal{M}_i$: ML model used by branch $c_i$; $y_i$: predictions produced by $c_i$ for version $v$
$v_i$: version tag attached to $y_i$; $v_{\text{seen}}$: last version processed by a given consumer
$v^* = \max(v_1, \ldots, v_n)$: latest version available across all branches
$\Delta_i = v^* - v_i$: per-branch staleness (version gap)
$P_t$: fused perception state at time $t$;
{— Producer process —}
$v \leftarrow 0$
**while** running **do**
  $f \leftarrow$ ReadCamera()
  WriteShared($f$, $v$)
  NotifyConsumers()
  **if** AtLeastOneConsumerStarted($v$) **then**
    {Partial-Blocking advance criterion} $v \leftarrow v + 1$
  **end if**
**end while**
{— Consumer process $c_i$ —}
$v_{\text{seen}} \leftarrow -1$
**while** running **do**
  $(v, f) \leftarrow$ ReadLatest() {Non-blocking: may return same $v$ as before}
  **if** $v \neq v_{\text{seen}}$ **then**
    $v_{\text{seen}} \leftarrow v$
    $y_i \leftarrow$ Infer($\mathcal{M}_i, f$)
    SignalStarted($v$)
    WriteResult($c_i$, $v$, $y_i$)
  **end if**
**end while**
{— Manager process —}
**while** running **do**
  $\{(v_i, y_i)\}_{i=1}^{n} \leftarrow$ CollectLatest($C$) {$v_i$: version from branch $i$, $y_i$: its predictions}
  $v^* \leftarrow \max(v_1, \ldots, v_n)$
  $\Delta_i \leftarrow v^* - v_i \quad \forall i \in \{1, \ldots, n\}$
  $P_t \leftarrow$ MergeLatest($y_1, \ldots, y_n$)
  ResolveCollisions($P_t$, *policy*) {e.g., confidence-first, IoU-NMS, or prefer newer $v_i$}
  Publish($P_t$, $v^*$, $\Delta$)
**end while**=0

**Behavior summary.** Under partial-blocking scheduling, the producer maintains constant frame flow while ensuring that every frame is processed by at least one consumer. Faster branches (e.g., lane detection) publish frequent updates, while slower ones catch up asynchronously. The manager merges

the most recent outputs from all perception pipelines, retaining the latest version available for each branch.

## 4.4 Worked Example (Trace)

To illustrate how the implemented partial-blocking scheduling behaves in practice, consider three consumer branches: $c_1$ (objects), $c_2$ (signs), and $c_3$ (lanes). Their respective average inference times are:

Objects: 12 ms Signs: 28 ms Lanes: 16 ms Capture: 10 ms

1. At $t$=0, the PRODUCER captures frame $f_{10}$ and writes it to shared memory with version $v$=10. All branches are notified and begin reading from the shared buffer.

2. The fastest branch, $c_1$ (objects), immediately signals that it has started processing version $v$=10. This satisfies the partial-blocking condition (*at least one consumer started*), allowing the producer to proceed and capture the next frame $f_{11}$ with $v$=11 after 10 ms.

3. Meanwhile, $c_1$ publishes its predictions ($v_1$=10, $y_1$=$O_{10}$) after 12 ms, $c_3$ (lanes) publishes ($v_3$=10, $y_3$=$L_{10}$) after 16 ms, and $c_2$ (signs), being the slowest, completes ($v_2$=10, $y_2$=$S_{10}$) only after 28 ms. By this time, the producer has already captured $f_{11}$ and possibly $f_{12}$, so the current global version counter is $v$=12.

4. $c_1$ finishes processing $f_{11}$ and publishes ($v_1$=11, $y_1$=$O_{11}$) before $c_2$ even finishes version 10. $c_3$ completes ($v_3$=11, $y_3$=$L_{11}$) shortly after. At this point, the manager collects the latest results from all branches:

$$(v_1, y_1) = (11, O_{11}), \quad (v_2, y_2) = (10, S_{10}), \quad (v_3, y_3) = (11, L_{11}).$$

5. The manager determines the newest version $v^*$=11 and computes the staleness vector $\Delta$=($v^*-v_1$, $v^*-v_2$, $v^*-v_3$) = (0, 1, 0). It merges the latest predictions using MERGELATEST($O_{11}, S_{10}, L_{11}$) and applies optionally applies collision handling where overlapping detections exist (e.g., by choosing the higher-confidence or newer result). The fused perception state $P_t$ is then published as the system's current output.

# Chapter 5

# Application (numerical validation)

Explain the experimental methodology and the numerical results obtained with your approach and the state of art approache(s).

Try to perform a comparison of several approaches.

Statistical validation of the results.

## 5.1 Methodology

- What are criteria you are using to evaluate your method?
- What specific hypotheses does your experiment test? Describe the experimental methodology that you used.
- What are the dependent and independent variables?
- What is the training/test data that was used, and why is it realistic or interesting? Exactly what performance data did you collect and how are you presenting and analyzing it? Comparisons to competing methods that address the same problem are particularly useful.

## 5.2 Data

Describe the used data.

## 5.3 Results

Present the quantitative results of your experiments. Graphical data presentation such as graphs and histograms are frequently better than tables. What are the basic differences revealed in the data. Are

they statistically significant?

## 5.4 Discussion

- Is your hypothesis supported?
- What conclusions do the results support about the strengths and weaknesses of your method compared to other methods?
- How can the results be explained in terms of the underlying properties of the algorithm and/or the data.

# Chapter 6

# SWOT Analysis

# Chapter 7

# Conclusion and future work

Try to emphasise the strengths and the weaknesses of your approach. What are the major shortcomings of your current method? For each shortcoming, propose additions or enhancements that would help overcome it.

Briefly summarize the important results and conclusions presented in the paper.

- What are the most important points illustrated by your work?
- How will your results improve future research and applications in the area?

# Chapter 8

# Latex examples

Item example:

- content of item1
- content of item2
- content of item3

Figure example

... (see Figure 8.1)


Figure 8.1: The evolution of the swarm size during the GA generations. This results were obtained for the $f_2$ test function with 5 dimensions.

Table example: (see Table 8.1)

Algorithm example

... (see Algorithm 2).

Table 8.1: The parameters of the PSO algorithm (the micro level algorithm) used to compute the fitness of a GA chromosome.

| **Parameter** | **Value** |
|---|---|
| Number of generations | 50 |
| Number of function evaluations/generation | 10 |
| Number of dimensions of the function to be optimized | 5 |
| Learning factor $c_1$ | 2 |
| Learning factor $c_2$ | 1.8 |
| Inertia weight | $0.5 + \frac{rand()}{2}$ |

**Algorithm 2** SGA - Spin based Genetic AQlgorithm

```
BEGIN
@ Randomly create the initial GA population.
@ Compute the fitness of each individual.
for i=1 TO NoOfGenerations do
  for j=1 TO PopulationSize do
    p ← RandomlySelectParticleFromGrid();
    n ← RandomlySelectParticleFromNeighbors(p);
    @ Crossover(p, n, off);
    @ Compute energy ΔH
    if ΔH satisfy the Ising condition then
      @ Replace(p,off);
    end if
  end for
end for
END =0
```