RP2.2
Automated Testing of Unmanned Aerial Vehicles

Author: **Giulio Montuori,** *Professor:* **Matteo Camilli,**

Politecnico di Milano

0 INTRODUCTION

This report presents a UAV test case generator developed for the SBFT-inspired R2.2 project. The tool is designed to find potential vulnerabilities in PX4's obstacle avoidance system by generating test cases using search-based technique. The generator integrates with Aerialist, a test bench built on top of PX4, and uses `NSGA-II` algorithm to explore configurations of static obstacles that might lead to UAV crashes or dangerously close encounters, also known as hard and soft fail in the **UAV Testing Competition**.

1 System Architecture

The system comprises two main components, `start.py` the test generator and `middle.py` the Aerialist docker execution daemon:

- `start.py`: Executes the optimization algorithm (`NSGA-II` via `pymoo`), generates test configurations, and encodes them as `JSON`. This runs on the latest available version of Python.
- `middle.py`: Continuously watches the shared folder, converts the `JSONs` into `YAML` test files, run the simulation via PX4 and Gazebo, and writes back the result in `JSONs`. This run on Pyhton 3.8.0 as required by Aerialist inside the docker container.

Each test cycle involves converting genomes to 3D obstacle placements, running the simulation, extracting flight trajectories, and computing fitness metrics. Inter-process communication is handled via a shared directory.

2 Genetic Encoding and Fitness Objectives

Each test case comprises three non-overlapping box-shaped obstacles, each defined by six genes: `[x, y, l, w, h, r]`. Thus, genomes are 18-dimensional vectors. The following objectives are optimized:

- Minimize distance to obstacles: Simulation returning UAV minimum distance, those under `1,5` meters are considered risky as defined in UAV Testing Competition.
- Maximize diversity: Novelty is quantified via DTW on 2D flight trajectories.
- Minimize flight duration: Shorter test are preferred.

2.1 Fitness Function and Penalization

Each individual genome is evaluated on three objectives, forming a vector $F(x) = [f_1(x), f_2(x), f_3(x)]$, where:

- $f_1(x)$ = minimum distance to obstacle:

$$f_1(x) = \begin{cases} 10^6, & \text{simulation fail} \\ 100 \cdot d_{\min}(x), & d_{\min}(x) > 0 \\ 0.1, & d_{\min}(x) = 0 \end{cases}$$

  where $d_{min}(x) = 0$ only happen in the unlucky cases there the UAV goes above an obstacle which should be possible if the obstacle hight is above 5 meters according to the offical FAQ of the UAV Testing Competition.
- $f_2(x)$ = negative novelty score, based on DTW distance to previous trajectories.
- $f_3(x)$ = flight duration, extracted from the UAV trajectory logs and converted from nanoseconds to seconds.

Penalization is applied if the obstacle configuration is invalid or if the simulation fails or times out. In both cases, all objectives are assigned the constant penalty to ensure these individuals are discarded during survival selection.

2.2 Trajectory Novelty Metric

To promote diversity among test cases, the generator computes a novelty score using Dynamic Time Warping (DTW) a sequence alignment technique used to measure the similarity between two time-dependent trajectories using dynamic programming. Given two sequences of 2D points $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_m)$ where $a_i, b_j \in \mathbb{R}^2$, DTW finds a warping path $W =$

$((i_1, j_1), ..., (i_K, j_K))$ that minimizes the total alignment cost:

$$DTW(A,B) = \min_{W} \sum_{(i,j)\in W} \|a_i - b_j\|_2$$

This allows comparison of trajectories even if they are not sampled at the same rate or do not progress uniformly in time. Each UAV trajectory is down-sampled and transformed into a sequence of `(x, y)` points. The novelty is defined as the DTW distance between the current trajectory and those stored in an archive of previously generated trajectories. Higher DTW values indicate greater novelty.

2.3 Constraints Enforcement

To ensure the generation of valid test cases, the following constraints are enforced on all obstacle configurations:

- Non-overlapping obstacles: This is enforced using geometric polygon intersection via the Python library `Shaply`.
- Ground placement: All obstacles are placed with a fixed vertical position $z = 0$, in accordance with competition rules.
- Parameter bounds: Each obstacle is defined by a 6-dimensional vector $[x, y, l, w, h, r]$ with the following domain bounds:

$$\begin{aligned} x &\in [-40m, 30m] && \text{(position X)} \\ y &\in [10m, 40m] && \text{(position Y)} \\ l &\in [2m, 20m] && \text{(length)} \\ w &\in [2m, 20m] && \text{(width)} \\ h &\in [10m, 25m] && \text{(height)} \\ r &\in [0^\circ, 90^\circ] && \text{(rotation angle)} \end{aligned}$$

All constants like number of obstacles, timeouts, population size, DTW sampling rate, and penalty values, are stored in a centralized configuration file `parameters.py`. This design enables rapid experimentation and configuration. Modifying this file allows the test generator to adapt to different mission plans, or optimization settings without changing core logic.

3 NSGA-II with custom Survival and Sampling

The algorithm uses a custom survival operator `ToleranntLexicoSurvival` to prioritize solutions via lexicographic ordering:

1. Distance: The main source of point.
2. Diversity: To avoid penalty.
3. Time.

Tolerances are applied to avoid over fitting to negligible differences and promote diversity. A diagnostic logger prints best distances, mean and max novelty and durations score per generation.

To improve the effectiveness of the initial population, a custom sampling strategy is used via `PathSeedSampling`. Each obstacle is placed near a randomly interpolated point along the predefined mission route build from waypoints in the available `mission.plan` file. This ensures that obstacles are placed near the UAV's supposed flight path, increasing the likelihood of meaningful interaction and accelerating convergence during early generations.

4 Simulation Pipeline

Simulations are executed in docker with the `middle.py` daemon that:

- Converts `JSON` test descriptions into `YAML`.
- Constructs `DroneTest` instances and runs them via Aerialist.
- Records trajectories, duration, and minimum distances.
- Applies a timeout fail safe in case of a test structured in a way where the UAV can't reach one of the way-points of the mission plan.

All results are saved as `JSON` files and fetched by the generation once simulations complete. The results includes timestamped trajectory of UAV, which is then parsed to compute the distance and DTW novelty. Simulations that exceed the maximum allowed duration are terminated, and their results are replaced with an error `JSON`. This prevents blocking the optimization loop and ensures robust handling of invalid or stalled tests.

5 Evaluation and Output

Each simulation produces:

- A `YAML` test file, used for the simulation and to build the final folder of top tests.
- A `JSON` result with minimum distance, time, and a timestamped flight path.

At the end of the optimization, the best test cases from all generation are selected using a tolerance filter and saved to a final directory.

6 Reproducibility and Configuration

Parameters such as obstacle bounds, evolution settings, and runtime controls are defined in a centralized file `parameters.py` for ease of tuning and experimentation. The generator ensures termination of docker container using Python's library `atexit` at the and of the execution. The generator is fully deterministic under the same random seed (`SEED=42`). It communicates with the Aerialist daemon via a shared volume between the Docker container and the Linux environment.