Potato: A Data-Oriented Programming 3D Simulator for Large-Scale Heterogeneous Swarm Robotics

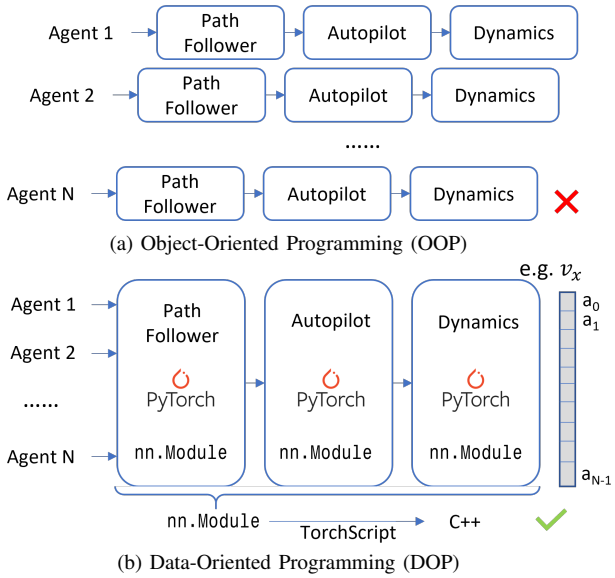
Jinjie Li[1], Liang Han[2*], Haoyang Yu[2], Zhaotian Wang[2], Pengzhi Yang[3], Ziwei Yan[2], Zhang Ren[1]

***Abstract*— Large-scale simulation with realistic nonlinear dynamic models is crucial for algorithms development for swarm robotics. However, existing platforms are mainly developed based on Object-Oriented Programming (OOP) and either use simple kinematic models to pursue a large number of simulating nodes or implement realistic dynamic models with limited simulating nodes. In this paper, we develop a simulator based on Data-Oriented Programming (DOP) that utilizes GPU parallel computing to achieve large-scale swarm robotic simulations. Specifically, we use a multi-process approach to simulate heterogeneous agents and leverage PyTorch with GPU to simulate homogeneous agents with a large number. We test our approach using a nonlinear quadrotor model and demonstrate that this DOP approach can maintain almost the same computational speed when quadrotors are less than 5,000. We also provide two examples to present the functionality of the platform.**

I. Introduction

Swarm robot systems can accomplish tasks that individual robots cannot complete alone through cooperation and coordination, which has recently received extensive attention from academia and industry. Developing perception, planning, and control algorithms for these swarm robots requires experiments on hardware systems to verify their effectiveness. However, field experiments for swarm robots are demanding to conduct due to challenges such as large experimental sites, high maintenance difficulty, and high failure rate. Therefore, utilizing a simulator with realistic models is necessary to verify the swarm algorithms.

The simulators suitable for swarm robots have requirements in two dimensions: the number of simulation nodes and the fidelity of the simulation models. However, existing robotic simulators usually focus on one dimension. Some robotic simulators attempt to achieve large-scale simulations at the expense of fidelity, such as the simple kinetic model in BeeGround [1] and the modified unicycle model in SCRIMMAGE [2]. Other popular robotic simulators provide realistic models while supporting the simulation of only about 50 robots on a desktop computer, such as AirSim [3] and Gazebo-based RotorS [4]. The above simulators leverage mainly central processing units (CPUs) for numerical computation, which limits their capability for large-scale simulation.



(a) Object-Oriented Programming (OOP)

(b) Data-Oriented Programming (DOP)

Fig. 1. The core idea of our simulator. Traditional robotic simulators are developed using OOP, where multiple agents are multiple instances as shown in (a). The agents are then computed through for-loops, multi-threads, or multi-processes, etc. However, since a desktop CPU typically has 10-20 threads, each CPU thread computes multiple agents serially in a loop for large-scale simulation. As a result, the computational speed increases almost linearly with the number of agents. In contrast, the proposed simulator is developed using DOP, grouping the computations of homogeneous agents together and parallelizing them in batches using tensors as shown in (b), which can be computed directly on GPUs. In addition, each computation module can be written as an `nn.Module` and compiled into C++ using PyTorch's TorchScript for further acceleration. This approach maintains almost the same computational speed for the number of agents below a certain level (below 5,000 in our test).

Alternatively, the advancement of graphics processing units (GPUs) has opened up the potential for conducting large-scale and high-quality simulations in parallel. NVIDIA's Isaac Gym [5] is an example simulation tool that utilizes GPUs to parallelly simulate the physical world, indicating that GPUs can handle nonlinear models with high fidelity for large-scale simulations. However, Isaac Gym primarily aims at the algorithm development of deep reinforcement learning, which provides an interface differing from the requirement of swarm robotics. In addition, Isaac Gym is developed using CUDA and C++, which is difficult to master and modify internally, making it challenging for scientific research on swarm algorithms. These limitations are considered when developing the proposed simulator.

In this paper, we develop **Potato**, a GPU-based large-scale

[1]J. Li and Z. Ren are with the School of Automation Science and Electrical Engineering, Beihang University, Beijing, 100191, China {lijinjie, renzhang}@buaa.edu.cn

[2]L. Han, H. Yu, Z. Wang, and Z. Yan are with the Sino-French Engineer School, Beihang University, Beijing, 100191, China {liang_han, haoyang_yu, wangzhaotian, yanziwei}@buaa.edu.cn

[3]P. Yang is with the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, 2628 CD, Netherlands P.Yang-4@student.tudelft.nl

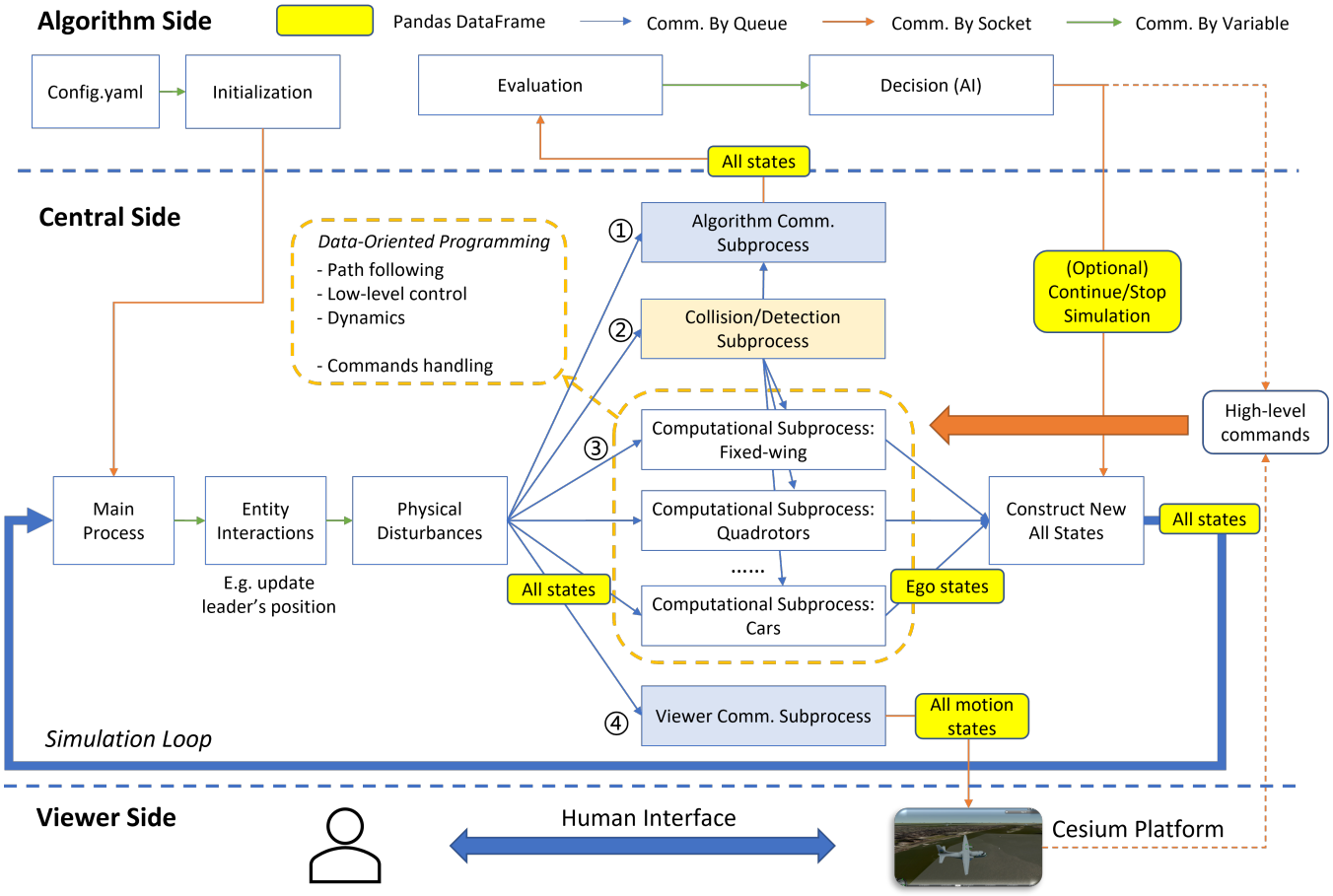


Fig. 2. The system structure of the proposed simulator. The proposed simulator consists of a Simulation Loop where the states of all agents are transmitted to four directions. Direction ① sends the states to the *algorithm side* via an algorithm communication subprocess, which uses this information for evaluation and decision making. The generated commands are then sent to computational subprocesses for handling. Direction ② calculates collision and detection results, which are also sent to computational subprocesses for handling. Direction ③ computes low-level algorithms and dynamics for heterogeneous agents, and sends the updated states back to the main process to refresh the all-states data. Finally, Direction ④ uses a viewer communication subprocess to visualize all agents' motions, and users can manipulate the mouse to influence the agents' behaviors.

heterogeneous robot simulator. Unlike traditional OOP-based simulators, Potato is designed using DOP, naturally supporting the numerical computation of large-scale nodes on GPUs. Compared with Isaac Gym, our platform is developed using Python and PyTorch and has cross-platform compatibility as well as lower development difficulty. Furthermore, the simulation can be accelerated using TorchScript, a tool in PyTorch used for deep learning acceleration. Finally, we conduct experiments to verify the effectiveness of the proposed architecture, and we present two demos based on this platform. We hope the idea proposed in this paper can promote the development of the next-generation swarm robotic simulator.

II. Methodology

A. System Architecture

This section introduces the system architecture of the developed large-scale heterogeneous simulation platform, as shown in Fig. 2. From top to bottom, the entire system is divided into three sides: an *algorithm side*, a *central side*, and a *viewer side*. The *algorithm side* mainly generates decision-making instructions according to the states of the agents; the *central side* controls the simulation process and computes the ordinary differential equations (ODEs) of dynamics; the *viewer side* displays the movement of agents and servers as a human-computer interface. Socket communication is used between each end, making these ends capable of running on different computers. Furthermore, the simulator written in Python can run on different operating systems.

The whole simulation process is depicted in Fig. 2. At the beginning, the *algorithm side* sets the simulation parameters by reading a configuration file. Then, during each simulation loop, all agents' states (stored in a `pandas.DataFrame`) are circulated in different system modules, and each module changes all or part of the agents' states.

Unlike other simulation platforms, we take the collision/detection module and *algorithm side* out of the main simulation loop based on the following considerations: First, keeping as few modules as possible in the main loop accelerates the computational speed. Second, taking the decision module (*algorithm side*) outside of the loop approximates the real world. In the real world, humans make decisions with the changing physical world, so the physical world can run for a while before receiving decision-making instructions.

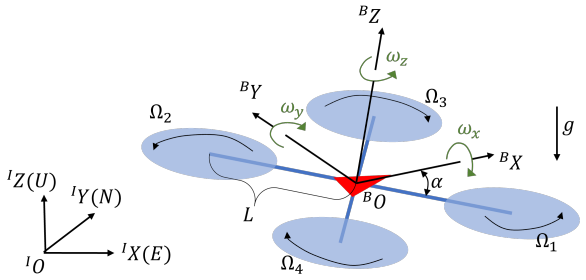


Fig. 3. Diagram of the quadrotor model with the ENU (X East, Y North, Z Up) inertial frame and the FLU (X Forward, Y Left, Z Up) body frame.

Third, taking the collision/detection module outside can still guarantee the correct result as long as its computational speed is similar to the main loop. Then the collision information is handled in the main loop using an event-triggered mechanism, and colliding agents will be marked as dead and excluded from the entire system. We also retain the option of putting these modules into the main loop.

B. Quadrotor Dynamics & Control

Three types of mobile robots have been implemented in this simulator, including fixed-wing drones [6], quadrotors [7], and cars [8]. The quadrotors are utilized to test the performance and hence are briefly introduced here.

We assume that the origin of the body frame $\mathcal{B}$ is at the center of mass, and four rotors are all placed in the $\mathcal{B}$ frame's XY-plane. Established from 6-DoF rigid-body dynamics, the quadrotor model is written as follows

$$^I\dot{\boldsymbol{p}} = {}^I\boldsymbol{v}, \tag{1}$$

$$^I\dot{\boldsymbol{v}} = \left({}^I_B\boldsymbol{R}(\boldsymbol{q}) \cdot {}^B\boldsymbol{f}_u\right)/m + {}^I\boldsymbol{g}, \tag{2}$$

$${}^I_B\dot{\boldsymbol{q}} = 1/2 \cdot {}^I_B\boldsymbol{q} \circ \begin{bmatrix} 0 \\ {}^B\boldsymbol{\omega} \end{bmatrix}, \tag{3}$$

$${}^B\dot{\boldsymbol{\omega}} = \boldsymbol{I}^{-1} \cdot \left(-{}^B\boldsymbol{\omega} \times \left(\boldsymbol{I} \cdot {}^B\boldsymbol{\omega}\right) + {}^B\boldsymbol{\tau}_u\right), \tag{4}$$

where $\circ$ indicates quaternion multiplication, m is mass, ${}^I\boldsymbol{g} = [0, 0, -g]^T$ is gravity vector, $\boldsymbol{I} = \texttt{diag}(I_{xx}, I_{yy}, I_{zz})$ is inertia matrix assuming that the quadrotor exhibits symmetry across all three axes, ${}^B\boldsymbol{f}_u$ and ${}^B\boldsymbol{\tau}_u$ are force and torque caused by the rotors, and ${}^B\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^T$ is angular rate vector expressed in the body frame.

The thrust generated by rotors is assumed to be vertical to the $\mathcal{B}$ frame's XY-plane, and we therefore obtain ${}^B\boldsymbol{f}_u = [0, 0, f_c]^T$ and ${}^B\boldsymbol{\tau}_u = [\tau_x, \tau_y, \tau_z]^T$, where f_c is the collective force of four rotors. We use a quadratic fit to model the thrust and torque for each propeller:

$$f_i = k_t \cdot \Omega^2, \quad \tau_i = k_q \cdot \Omega^2, \tag{5}$$

where k_t and k_q are the thrust coefficient and torque coefficient, respectively, as well as Ω represents motor speed in RPM. Then the $[f_c, \tau_x, \tau_y, \tau_z]^T$ and the thrust of each rotor f_i is connected by

$$[f_c, \tau_x, \tau_y, \tau_z]^T = \boldsymbol{G} \cdot [f_1, f_2, f_3, f_4]^T, \tag{6}$$

TABLE I

RUNNING TIME MEAN AND STANDARD DEVIATION (SD) PER ROUND

Language Version	PyTorch Version	FALLBACK Error	Running Time [ms] (mean ± SD)
origin Py3.8	1.10.0+cu102	N	**0.8452** ± 0.0196
conda Py3.8	1.10.0+cu102	N	0.8550 ± 0.0127
conda Py3.9	1.13.0+cu116	Y	1.6974 ± 0.0141
conda Py3.9	2.0.0+cu117	N	1.4171 ± 0.0187
conda Py3.10	1.12.0+cu116	Y	1.1738 ± 0.0248
conda Py3.9	1.12.0+cu116	Y	1.1857 ± 0.0071
conda Py3.8	1.12.0+cu116	N	0.8678 ± 0.0081
C++ Release	1.12.0+cu116	Y (much)	2.8181 ± 0.0460
C++ Debug	1.12.0+cu116	Y (much)	2.8017 ± 0.0318

in which the control allocation matrix $\boldsymbol{G}$ is

$$\boldsymbol{G} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ L\sin\alpha & -L\sin\alpha & -L\sin\alpha & L\sin\alpha \\ -L\cos\alpha & -L\cos\alpha & L\cos\alpha & L\cos\alpha \\ k_q/k_t & -k_q/k_t & k_q/k_t & -k_q/k_t \end{bmatrix}, \tag{7}$$

where L and α are geometric parameters shown in Fig. 3. Finally, the model is discretized by the 4-order *Runge-Kutta* method for numerical simulation.

We also implement a PID body rate controller as the inner control loop. This dynamics & control model is utilized to test the computational speed in the next section, and we recommend [7] to interested readers for more details.

III. PERFORMANCE

In this section, we test the computational performance of the DOP structure on simulating swarm quadrotors. The performance is tested using a desktop computer with an Intel i7-10700 CPU and an NVIDIA GTX 1660 SUPER GPU. The test program runs on the Ubuntu 20.04 operating system.

The proposed method relies on TorchScript provided by PyTorch to accelerate the computational speed, and the whole simulation loop can be implemented by Python or C++, so we test the running time for each round under different languages and PyTorch versions. In each test, we fix the number of quadrotors to 1,000 and first run 500 rounds for stable running, then run 2,000 rounds and calculate the average consuming time. Finally, we execute three times for each test and list the result in Table I.

From the table, the C++ version is not faster than the Python version, and we infer the possible reason is many FALLBACK warnings when loading the TorchScript model using `libtorch`, the C++ API of PyTorch.

Then we choose the conda Python 3.8 environment with PyTorch 1.12.0+cu116 to test the change of running time concerning the number of quadrotors, as shown in Fig. 4. The figure shows that the computational time stays almost the same under 5,000 agents and remains less than 2ms for even 10,000 agents, demonstrating the advantage of the DOP method in simulating large-scale agents.

IV. EXAMPLES AND EXTENSIONS

This section presents two demos of our simulator.

The first demo (Fig. 5a) verifies that our simulator can support over 1000 homogeneous agents simulating on one