B(E)3M33MRS - Task 03 Swarm

Pavel Petráček

November 23, 2022

v1.0

Contents

1 Introduction

The third task focuses on sensor-driven **decentralized** control and decision making in multi-agent systems. The goal of the task is similar to the objective of `task_02_formation`; however, the architecture will differ — you will be writing programs which will be run many times in parallel with the same parametrization, once for each agent in the world.

2 Requirements

- Task **T.1**: Implement a decentralized multi-agent control and value-agreement strategy.
  - Program the contents of the `updateAgentState()` function in `task_03_swarm:packages/swarm/src/boids.cpp`.
- Task **T.2**: Implement a non-increasing function for weighting of vectors separating an agent from others and from obstacles.
  - Program the contents of the `weightingFunction()` function in `task_03_swarm:packages/swarm/src/swarm.cpp`.
- Task **T.2**: Implement a decentralized multi-agent control strategy for finding a mobile source of signal.
  - Program the contents of the `updateAction()` function in `task_03_swarm:packages/swarm/src/swarm.cpp`.

The red-labeled circle tasks are verified automatically using the Brute upload system.

3 Preliminaries

Decentralized formation control is a multi-agent control scheme that cannot be implemented using a centralized element which would act on the base of availability of all the required data. Instead, the control scheme is to be written such that both local and global cooperation is achieved with respect to the current or short history of past sensory observations. These sensory observations are available only locally, are subject to outages (e.g., due to visual occlusions), and contain stochastic noise. However, studies on swarming theory (hot research topic) have shown that a decentralized multi-robot system can use this deprived information to globally behave as a cooperative group — a self-organizing behavior emerging from local information and actions only. To achieve such a behavior, we need to introduce two building blocks — sensor-driven control and the ability to agree on a common goal with partial and inaccurate information.

3.1 Local-information swarming

In nature, there exist biological systems capable of forming groups comprising of rather simple agents. Although these agents usually have highly limited sensory inputs, these systems are capable of both local and global behaviors scalable up to thousands of agents (e.g., bird flocks and fish schools). The research of these systems has shown that the behavior is self-organizing and may be emerging from local actions arising from both the interoceptive and exteroceptive information. In natural systems, these actions (or behaviors) represent the need for finding food, water and shelter, scouting and avoiding predators, socialization, mating, migration, hibernation, and even fun. Species to species, these behaviors and their spectra differ and are subject to different priorities based on the domain, social habits and structures, position of the species in the food chain, period of the year, and many others.

The primary branch of research on swarming behaviors models these behaviors as multi-variable functions and utilizes them to act reactively with respect to the sensory input, typically only to the most recent one. These models combine the functions in order to derive the physical action (ideally optimal) to be performed in the next iteration. To make the models scalable, the amount of interactions is locally limited. This is done primarily by one of the two approaches — topological and metric. In the topological sense, the amount of exteroceptive information is limited to a finite number of nearest agents (e.g., field studies show that *starlings* act on the basis of six or seven animals surrounding it). In the metric sense, the agent acts on the basis of all agents within a fixed-size neighborhood, typically modeled as an agent-centered sphere. In this assignment, we'll be considering the metric approach. Common attribute of these

models is that they define a next-iteration action $\mathbf{a}(\cdot) \in \mathbb{R}^M$ as a linear combination of N aggregated functions (rules, behaviors) represented as $\mathbf{f}(\cdot) \in \mathbb{R}^{N \times M}$

$$\mathbf{a}(\cdot) = \mathbf{f}(\cdot)\, \boldsymbol{\omega}(\cdot) = [\mathbf{f}_1(\cdot), \mathbf{f}_2(\cdot), \ldots, \mathbf{f}_N(\cdot)]\, [\omega_1(\cdot), \omega_2(\cdot), \ldots, \omega_N(\cdot)]^T, \tag{1}$$

where M is the domain dimension, $\mathbf{f}_i(\cdot) \in \mathbb{R}^M$, $i \in 1, \ldots, N$, is a column vector and $\boldsymbol{\omega}(\cdot) \in \mathbb{R}^N$ is a rule-weighting column vector of scalar functions.

3.1.1 The Boids model

One of the oldest, simplest, and most utilized models combining several functions into a reactive action is the Boids model. For each neighbor within an agent neighborhood, the Boids model combines the following three rules into a final acting force:

- Velocity alignment: move in the same direction as your neighbors.
- Cohesion: stay close to your neighbors.
- Separation: avoid collisions with your neighbors.

To limit the amount of neighbors, the Boids model limits the perception in a metric fashion, sometimes with a different perceptual radius for each of the rules.

Given a set $\mathcal{N} = \{(\mathbf{p}_i, \mathbf{v}_i)\}$ of neighbors of an arbitrary finite size $|\mathcal{N}| \ll \infty$, where each neighbor i is specified by its position $\mathbf{p}_i \in \mathbb{R}^3$ and velocity $\mathbf{v}_i \in \mathbb{R}^3$ in the 3D space, the Boids model produces for an agent at position $\mathbf{p}$ an action $\mathbf{a}(\mathbf{p}, \mathbf{v}, \mathcal{N}) \in \mathbb{R}^3$. With relation to (1), this action is given as

$$\mathbf{a}(\mathbf{p}, \mathbf{v}, \mathcal{N}) = [\mathbf{f}_a(\mathbf{v}, \mathcal{N}), \mathbf{f}_c(\mathbf{p}, \mathcal{N}), \mathbf{f}_s(\mathbf{p}, \mathcal{N})]\, [\omega_a, \omega_c, \omega_s]^T, \tag{2}$$

where

$$\mathbf{f}_a(\mathbf{v}, \mathcal{N}) = \frac{1}{|\mathcal{N}| + 1} \left(\mathbf{v} + \sum_{i=1}^{|\mathcal{N}|} \mathbf{v}_i \right), \qquad \mathbf{f}_c(\mathbf{p}, \mathcal{N}) = \frac{1}{|\mathcal{N}|} \sum_{i=1}^{|\mathcal{N}|} \mathbf{p}_i - \mathbf{p}, \qquad \mathbf{f}_s(\mathbf{p}, \mathcal{N}) = \frac{1}{|\mathcal{N}|} \sum_{i=1}^{|\mathcal{N}|} \mathbf{p} - \mathbf{p}_i, \tag{3}$$

represent the alignment, cohesion, and separation rules; and $\boldsymbol{\omega} = [\omega_a, \omega_c, \omega_s]$ represents scalar weighting coefficients. In this notation[1], the cohesion and separation rules force an agent to stay close to and separate from the weighted geometrical center of the neighbors set $\mathcal{N}$. Although very simple, the Boids model produces self-organizing behavior with its properties being directly related to the vector $\boldsymbol{\omega}$. If no noise is present in the observations, the model converges to an equilibrium of forces in a finite number of iterations. In the equilibria, the distance between the agents relates directly to the ratios of the coefficients in the $\boldsymbol{\omega}$ parametrization.

The Boids model validated that a set of decoupled rules can be combined together to reasonably mimic the behavior observed in biological systems. Because it was the first model proposing such a methodology, it is the baseline for further and more elaborate models combining a set of decoupled rules.

In the task **T.1** of this assignment, you will be utilizing this model and selecting a feasible vector $\boldsymbol{\omega}$ in order to achieve collective navigation in a simplistic world. In the task **T.2** of this assignment, you may utilize the model partially or entirely, but you will have to design your own rules to succeed, particularly for navigation through the environment.

3.2 Swarming with physical agents between obstacles

3.2.1 Rules weighting

The swarming models aggregating the decoupled behaviors according to (1) are designed by default for virtual dimensionless particles. In the original models, the particles are not compensated for their overlaps and thus collisions among the particles may occur. This makes these models unfeasible for use on real agents. To assure collision-free behavior of the system, we introduce non-linear distance-based weighting to the rules handling the separation of the agents. Such a function is arbitrary, but has a clear functionality — prevent collisions between two physical or virtual Unmanned Aerial Vehicles (UAVs) by scaling high with low mutual distance and vice versa. Example of such a function is shown in Figure 1. One of the objectives of task **T.2** is to design such a non-increasing function (does not have to be exactly the one shown in Figure 1) which will lead to safety and stability of your formation.

[1] Note that the equations are derived from the linguistic description and their exact formulation is not set.

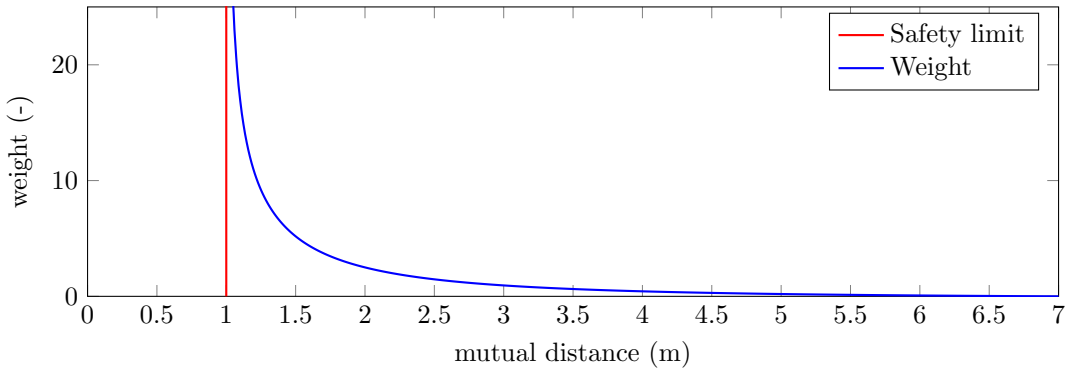

Figure 1: Example weighting as a function of mutual distance between two agents.

With the weighting function being related to the agent-to-agent distance, we may rewrite (2) to

$$\mathbf{a}(\mathbf{p},\mathbf{v},\mathcal{N}) = \left[\mathbf{f}_a(\mathbf{v},\mathcal{N}),\mathbf{f}_c(\mathbf{p},\mathcal{N}),\mathbf{f}_s(\mathbf{p},\mathcal{N})\right]\left[\omega_a,\omega_c,\omega_s(\mathbf{p},\mathcal{N})\right]^T. \tag{4}$$

3.2.2 Obstacle avoidance

The swarming models have been extended with variety of mechanisms to avoid obstacles. One of these mechanisms introduces the concept of *virtual* agents. A *virtual* agent represents, in some arbitrary way, either single obstacle or several obstacles at once. Similarly to a *physical* agent, a *virtual* agent is perceivable only in a local neighborhood and is characterized by a state. The state characterization of *virtual* agents allows us to easily utilize the same (or at least very similar) behaviors defined for the *physical* agent-to-agent cooperation (e.g., the Boids rules) for the *virtual* agents and hence to avoid obstacles with minimal overhead (i.e., implementation, computation).

To showcase an example, Figure 2 shows the *virtual* agents derived for circular and line-segment obstacles. In this example, the states of the *virtual* agents for agent $\mathcal{A}$ with *virtual* position $\mathbf{p}$ and *virtual* velocity $\mathbf{v}$ have been obtained as follows.

For **circular** obstacles, the *virtual* agents have *virtual* position equal to the closest point on the circle to agent $\mathcal{A}$ and *virtual* velocity defined as

$$\mathbf{p}_i = \mathbf{p} + \left(1 - \frac{r_i}{||\mathbf{c}_i||_2}\right)\mathbf{c}_i, \qquad \mathbf{v}_i = \frac{r_i}{||\mathbf{c}_i||_2}\left(\mathbf{I} - \frac{\mathbf{c}_i\mathbf{c}_i^T}{||\mathbf{c}_i||_2^2}\right)\mathbf{v}, \tag{5}$$

where r_i is the radius and $\mathbf{c}_i$ is the circle center of a circular obstacle i in frame of agent $\mathcal{A}$, and $\mathbf{I}$ is the identity matrix.

For **line-segment** obstacles, the *virtual* agents have *virtual* position $\mathbf{p}_i$ equal to the closest point on the line segment to agent $\mathcal{A}$ and *virtual* velocity given as

$$\mathbf{v}_i = \frac{1}{||\mathbf{p}_i||_2}\left(\mathbf{I} - \mathbf{n}_i\mathbf{n}_i^T\right)\mathbf{v}, \tag{6}$$

where $\mathbf{n}_i$ is the unit normal vector of the line segment.

It is clear from Figure 2 that the *virtual* position lies on the edge of an obstacle and the *virtual* velocity is a distance-scaled vector steering the movement along the respective obstacle. The definition in (5) and (6) assures that the vectors $\mathbf{v}_i$ lie in the same half-space of the space as $\mathbf{v}$, where the space is divided by a hyperplane $\mathbf{p} + t(\mathbf{p}_i - \mathbf{p})$, $t \in \mathrm{R}$.

Part of the objective of task **T.2** is to navigate a set of UAVs through a structured obstacle-filled environment. To do so, you may exploit the concept of *virtual* agents, e.g., by redefining (2) to

$$\mathbf{a}(\mathbf{p},\mathbf{v},\mathcal{N},\mathcal{O}) = \left[\mathbf{f}_a(\mathbf{v},\mathcal{N},\mathcal{O}),\mathbf{f}_c(\mathbf{p},\mathcal{N}),\mathbf{f}_s(\mathbf{p},\mathcal{N},\mathcal{O})\right]\left[\omega_a,\omega_c,\omega_s(\mathbf{p},\mathcal{N},\mathcal{O})\right]^T. \tag{7}$$

Remark 1. *The environment in task **T.2** does not have circular nor line-segment obstacles — you have to design the states of your virtual agents on the basis of the environment structure and available data.*

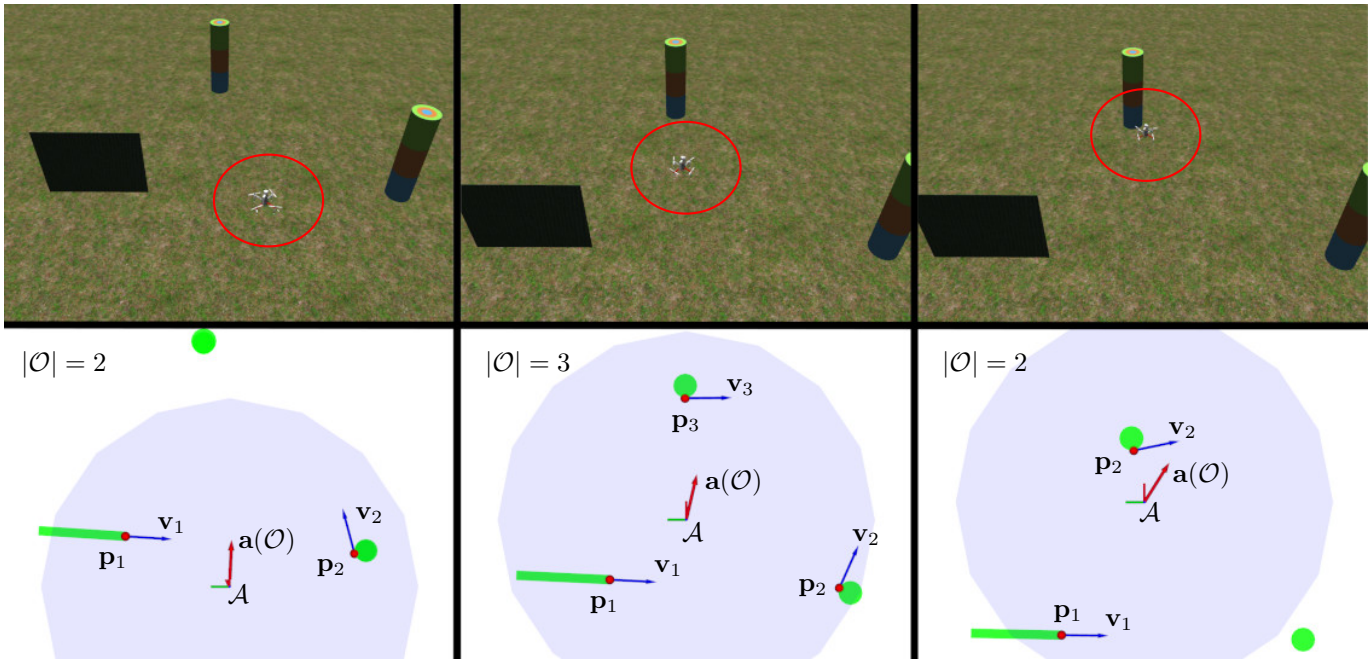

Figure 2: Example of *virtual* agents in an environment filled with circular and line-segment obstacles. The obstacles (the set $\mathcal{O}$) are perceived within a perception radius (light blue circle), with one *virtual* agent representing one obstacle. The state of the *virtual* agents is comprised of a position $\mathbf{p}_i$ (red dots) and a velocity $\mathbf{v}_i$ (blue arrows), $i \in 1, 2, 3$, in the body frame of agent $\mathcal{A}$. The output action $\mathbf{a}(\mathcal{O})$ (red arrow) is the result of applying an upwards-steering navigation and the Boids rules, as defined in (7) and weighted according to Figure 1, on the set of obstacles $\mathcal{O}$.

3.3 Consensus in distributed systems

Achieving consensus within a decentralized multi-agent system is about coming to an agreement about a value of a single or multiple variables among several distributed nodes in the presence of a number of faulty or uninformed agents. Some of the nodes/agents may fail or be unreliable in other ways, hence the consensus protocols must be robust to failures. There are plethora of models tackling consensus in various systems with different sources of failures, architectures, and properties. These models tackle the following selected problems:

- dimensions of the value to be agreed on — single value (e.g., Paxos protocol), binary consensus, and multiple values (e.g., Multi-Paxos and Raft protocols),
- failure types — crash (an agent abruptly stops and does not start working again) or a Byzantine crash (unspecified failure of an arbitrary type — e.g., sending contradictory/conflicting information or pausing activity for an arbitrary time period),
- time synchronization — synchronous and asynchronous systems,
- message authentication — required, allowed, or none,
- network topology — fully connected graphs, rings, trees, dynamic networks (agents come and leave the network arbitrarily),
- uniformity — non-uniform (agents may decide on different values) and uniform (no two agents are allowed to decide on different values).

Tackling all these problems is out of our scope of this task, so we will narrow our focus to the architecture of our tasks presented in section 4 and section 5. Our objectives require to uniformly agree on a single value in an asynchronous dynamic network where the failures can be modeled as a Byzantine crash. This combination of attributes makes the consensus probably the hardest to be achieved, yet required in a dynamic multi-agent system. It is tackled by two main concepts — **voting** on a value or a **majority selection** — based on the current available information.

In the voting scheme, an agent can propose a value and the receiving agents may or may not decide to accept it. A common approach in this scheme is to not propose a value but rather propose on becoming a leader of the group. Once the agents decide on a leader, the leader decides on the value of the entire system. However, problem of this concept

is how to decide on the leader and how to make sure the leader's decision is correct. In an unconnected multi-agent system, there may also be multiple leaders, which yields further problems in selecting which leader is superior if two or more leaders meet.

In the majority selection scheme, each agent decides on its value with respect to the current or a history of received information. This seems to go well with definition of swarming and with the objectives of our tasks, but this concept is sensitive to errors and inaccuracies. Implementing a majority selection must be robust to probabilistic errors. One way to do so in a single-value consensus over a fixed-domain (our case in **T.1** where the agents are agreeing on a single color out of four possible) is to represent the value with a probabilistic distribution and fusing the available information using the Bayes filter, such as

$$\bar{p}_k(\mathcal{C}_i) = \frac{p(Z|\mathcal{C}_i)\bar{p}_{k-1}(\mathcal{C}_i)}{\eta_k} \tag{8}$$

for deciding the belief (differentiated from probability by the overline $\bar{p}$) of state $\mathcal{C}_i$, $i \in 1, \ldots, M$, at time step k out of M discrete values given observations Z. Variable η_k represents a normalization constant ensuring that $\sum_{i=1}^{M} \bar{p}_k(\mathcal{C}_i) = 1$. Deciding on the value of an agent based on the probability distribution generated with the use of Bayes filter lowers the negative effects of stochastic noise in the measurements. Note that the equation above uses only the sensor model and misses the action model typical for Bayes filters in mobile robotics.

Remark 2. *If $\bar{p}(\mathcal{C}_i) = 1$, then $\bar{p}(\mathcal{C}_j) = 0, j \in 1, \ldots, M, j \neq i$, since $\sum_{i=1}^{M} \bar{p}(\mathcal{C}_i) = 1$. This makes $\bar{p}(\mathcal{C}_i)$ dominant and makes the value of $p(Z|\mathcal{C}_j)$ irrelevant in future iterations of the Bayes filter. Once $\bar{p}(\mathcal{C}_i) = 0$, it will never be nonzero again, no matter the acquired measurements. Because of this, it is a common practice to clamp the values of a discrete probability distribution to a certain interval such that $0 < p_{min} \leq \bar{p}(\mathcal{C}_i) \leq p_{max} < 1, i \in 1, \ldots, M$, and $\sum_{i=1}^{M} \bar{p}(\mathcal{C}_i) = 1$. This is particularly important in a dynamic world where the distribution changes over time, as is the case of our task.*

4 Task T.1: Boids

4.1 Description

The task focuses on implementing self-organizing navigation behavior scalable up to an arbitrary amount of agents. The behavior requires the agents to be capable of a collective decision making, i.e., to reach consensus on a common value. The control approach will be run N times in N independent synchronous instances with **identical parametrization**.

The task **environment** is bounded by rectangular bounds outside of which lives a terribly frightening, yet invisible predator. Four static beacons are located inside the rectangle, each defined by two distinct colors — color of the beacon itself $\mathcal{C}$ and color of the beacon it points to $\mathcal{C}^{\rightarrow}$. Inside the bounds live the agents trying to agree on what color should they all possess. Both the beacons and agents can be only of four distinct colors: {red, green, blue, orange}. Colors of the beacons and initial colors of the agents are randomized during every run of the task. An example **environment** of the task is shown in Figure 3.

The **agents** have the following properties:

- The agents are modeled as differential wheel drives with the forward velocity constrained to a strictly positive interval.
- Movement of the agents is limited to 2D.
- The agents know their own color and see position, velocity, and color of the agents in a constant-radius circle-like neighborhood.
- No direct communication among the agents is possible.
- The agents have no memory. They have to work with only the current available information.
- In each simulation iteration outside the `initialization` phase (see below), the agents can die (they are immortal during the `initialization` phase) if one of these scenarios happens:
  1. If two or more agents hit each other (are 0.3 m close or less), the weakest (a hidden state) one of them dies from the collision with probability of 30 %.

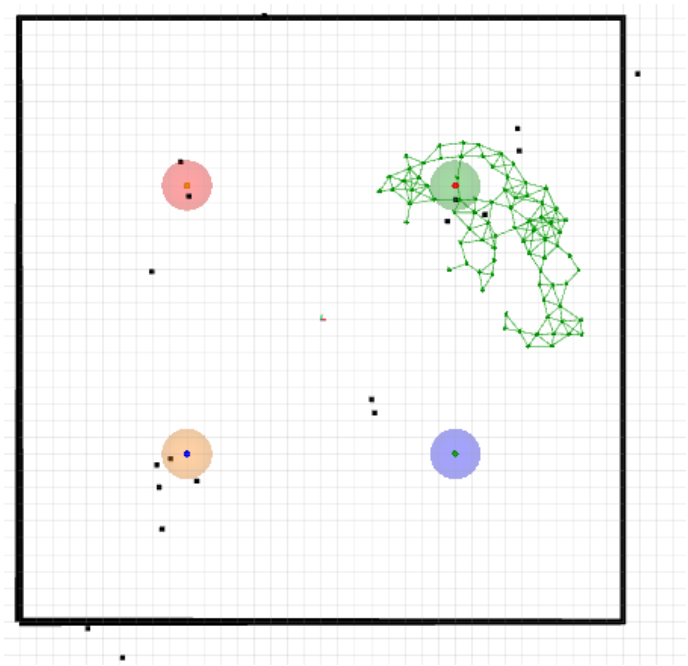

Figure 3: Example world of task **T.1**. The **black rectangle** defines the world bound. Inside the rectangle, the four colored circles denote beacons, each defined by two colors — inner color is its own color, outer color is the color of the beacon it points to. The agents are the smaller circles, here all in green. The agents which see each other are connected via colored edges. The beacons and the agents can yield only the colors {red, green, blue, orange} whereas the inactive parts of the environment are colored in **black**. Dead agents are represented by **black squares**. Beacons, world bounds, nor dead agents can move.

2. If an agent moves outside the world bound, it dies with probability of 1 % by terrible death of being eaten by the invisible, yet terribly frightening and always hungry predator.
3. If an agent does not have at least 2 other agents in radius of 5 m, it dies with probability of 20 % by loneliness.

- Agents distinguish colors of other nearby agents perfectly with 0 % error rate.
- When an agent is within the outer radius of a beacon, it may see the color $\mathcal{C}^{\rightarrow}$ the beacon points to. However, the agents are very bad in distinguishing colors of the beacons and may randomly classify $\mathcal{C}^{\rightarrow}$ as one of the three remaining colors with probability of 50 %. Although the agents are generally bad in distinguishing beacons' colors, 80 % of the agents were only taught to distinguish colors of other agents — these agents cannot distinguish colors of the beacons at all. However, the agents are socially responsible, thus if an agent sensitive to beacon colors is dying, the last thing it does is that it teaches a random living beacon-colorblind agent how to distinguish beacon colors, even if its just with 50 % success rate in distinguishing them.

The **task** goes as follows.

- A set of N agents is randomly (random position, velocity, and color distribution) spawned within the environment. This starts the `initialization` phase.
- The agents move through the environment. Once an agent sees a nearby agent, it detects its state (position, velocity, color) and can act on it by changing its own color and velocity.
- Once all the agents reach a consensus agreement $\mathcal{C}_0$ (all of them agree on the same color), the `initialization` phase ends and the agents are in danger of being killed from now on. **The value of $\mathcal{C}_0$ is arbitrary.** However, the first consensus agreement $\mathcal{C}_0$ defines the expected behavior for the rest of the task.
- Every time the agents reach a consensus agreement to color $\mathcal{C}_i$, they start receiving information about position of beacon with color $\mathcal{C}_i$. If the agents break the agreement (some agent changes its color), all the agents still keep receiving the last position related to beacon with color $\mathcal{C}_i$ till another agreement is reached.
- The agents are expected to agree on a common value in a correct order at least 5 times. The first agreed value $\mathcal{C}_0$

and colors of the beacons define the expected sequence of the agreed values. The expected sequence is given as $\{\mathcal{C}_0 \to \mathcal{C}_0^{\to} = \mathcal{C}_1 \to \mathcal{C}_1^{\to} = \mathcal{C}_2 \to \mathcal{C}_2^{\to} = \mathcal{C}_3 \to \mathcal{C}_3^{\to} = \mathcal{C}_0\}$. In the example shown in Figure 4, $\mathcal{C}_0$ is blue which yields the expected consensus-agreement sequence of {blue → orange → red → green → blue}.

- The task **succeeds** if the agents agree on the color sequence in the correct order with at least 70 % of surviving agents in 3600 iterations from the `initialization` start. Otherwise, the task **fails**.

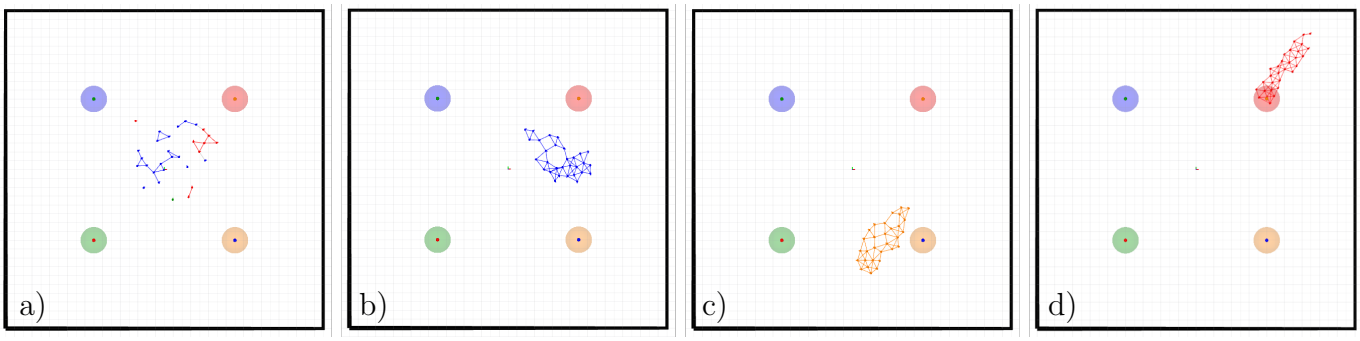

Figure 4: Example behavior of agents in task **T.1**. (a) The agents are spawned randomly (random position, velocity, and color) in the environment. (b) The agents agree on a blue color and start sensing position of the blue beacon (the bottom right). (c) After navigating to close proximity of the blue beacon, the beacon-color-sensitive agents have seen the beacon emits orange and have persuaded the rest to change their color to orange too. After they all agree on becoming orange, they start sensing the orange beacon (the top right). (d) After navigating to close proximity of the orange beacon, the beacon-color-sensitive agents have seen the beacon emits red and have persuaded the rest to change their color to red too. The expected behavior would continue this pattern till they all became blue again.

4.2 Perceptual inputs

The control of an agent $\mathcal{A}$ has the following perceptual inputs:

- Velocity of $\mathcal{A}$ in the world frame.
- Vector towards the beacon with color agreed on during last consensus agreement. During the `initialization` phase, vector towards (0, 0) is given instead. The Euclidean norm of the vector yields distance to the beacon.
- Discrete probability distribution representing the $\mathcal{A}$'s certainty about each of the colors.
  - The domain {red, green, blue, orange} is represented with a discrete probabilistic distribution over four possible values $P = \{p_r, p_g, p_b, p_o\}$, where $p_r + p_g + p_b + p_o = 1$.
  - A uniform color distribution is represented as $\{\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\}$.
  - An agent with distribution $\{1, 0, 0, 0\}$ is 100 % sure it should be red.
- Information whether $\mathcal{A}$ is nearby a beacon. If yes, the agent $\mathcal{A}$ disposes with the information about color $\mathcal{C}^{\to}$ of the beacon. Beware that in 50 % of cases, the agent distinguishes the color $\mathcal{C}^{\to}$ incorrectly. In other words, a beacon with $\mathcal{C}^{\to}$ = red = $\{1, 0, 0, 0\}$ is distinguished by the agent
  - as $\mathcal{C}^{\to} = \{1, 0, 0, 0\}$ with probability of $\frac{1}{2}$,
  - as $\mathcal{C}^{\to} = \{0, 1, 0, 0\}$ with probability of $\frac{1}{6}$,
  - as $\mathcal{C}^{\to} = \{0, 0, 1, 0\}$ with probability of $\frac{1}{6}$,
  - as $\mathcal{C}^{\to} = \{0, 0, 0, 1\}$ with probability of $\frac{1}{6}$.

  The same applies for all the colors from the domain, not only red.
- States of agents within the neighborhood $\mathcal{N}$ of agent $\mathcal{A}$:
  - position,
  - velocity in the world frame, and
  - discrete probability distribution representing the neighbor's certainty about each of the colors.

If not specified otherwise, the information is given in the body frame of $\mathcal{A}$. Beware that because only local information is available (except of the target vector), the agents have no notion about their global position in the world.

4.3 Implementation

All the *logic* is supposed to be implemented within the `task_03_swarm:packages/swarm/src/boids.cpp` file by filling function `updateAgentState()`.

```
/**
 * @brief Calculate a next-iteration action of one agent given relative information of its neighbors and the direction towards a target. This method is supposed to be filled in by the student.
 *
 * @param AgentState_t Current state of the agent as defined in agent_state.h.
 * @param user_params user-controllable parameters
 * @param action_handlers functions for visualization
 *  - visualizeArrow() will publish the given arrow in the agent frame within the visualization
 *
 * @return
 *    1) XYZ vector in frame of the agent to be set as velocity command. Beware that i) the vector z-axis component will be set to 0, ii) the vector magnitude will be clamped into <v_min, v_max> limits and iii) azimuth of the vector's XY-projection will be saturated such that the azimuth between the agent's current velocity and the vector does not exceed a maximal change.

 *       Example 1: Maximal change is d=45deg, v_min=0.1, v_max=0.2, and current velocity is (0.2, 0, 0) -> vector (0, 1, 1) will be clamped and saturated to 0.2*(cos(d), sin(d), 0).
 *       Example 2: Maximal change is d=45deg, v_min=0.1, v_max=0.2, and current velocity is (0.2, 0, 0) -> vector (0, -0.05, 1) will be clamped and saturated to 0.1*(cos(-d), sin(-d), 0).

 *    2) Probability distribution of colors to be set to the agent for next iteration. Beware that output distribution.dim() has to equal input state.distribution.dim().
 */
std::tuple<Eigen::Vector3d, Distribution> updateAgentState(const AgentState_t &state, const UserParams_t &user_params, const ActionHandlers_t &action_handlers);
```

This function will be called at the rate of 30 Hz since the moment all the agents are spawned in the virtual world. The function is pre-filled to assist you with tackling the task.

The implementation process can be divided into the following fundamental steps:

- Implement the Boids model and tune it without solving the value agreement yet.
- Once the agents produce collective behavior, implement value agreement.

4.4 Tips

To use (8) in our task for agreeing on values **among agents** requires a feasible sensor model which needs the knowledge of **hidden** joint probabilities (i.e., what is the probability of measuring the current observations given each possible value). Try to dynamically approximate the hidden probabilities (e.g., by values derived from the current perceptual inputs) and use the approximations in the value-agreement fusion among the agents.

Remember that the 20 % minority of agents that is able to distinguish color of beacons has to convince the remaining 80 % agents. This makes the concept of simple voting and majority selection insufficient for the assigned task. In our task, think whether it would be good to

- remain uncertain about a single color of an agent's color distribution, or to
- react to changes in local disagreement rather than agreement (counter-intuitive to the concept of majority selection).

The joint distribution for the **beacons** is known and using the Bayes formula

$$p(\mathcal{C}_i|Z) = \frac{p(Z|\mathcal{C}_i)p(\mathcal{C}_i)}{\sum_{j=1}^{M} p(Z|\mathcal{C}_j)p(\mathcal{C}_j)} \tag{9}$$

is possible and recommended since it will help you deal with the faulty measurements of the beacons' colors.

4.5 Testing, tuning, and validation

To **tune** your solution, use parameters passed to the `updateAgentState()` method via the `user_params` parameter. You may change default values of these parameters in `task_03_swarm:packages/swarm/config/user_params_boids.yaml` or update them on the run in the dynamic reconfigure window (do not forget to store your results in `task_03_swarm:packages/swarm/config/user_params_boids.yaml` after).

Implementation of the `updateAgentState()` can be **tested** by running simulation test `./simulation/run_boids_testing.sh VARIANT`, where `VARIANT` is one of {`testing`, `easy`, `medium`, `difficult`}. This will run your implementation once. In `testing` variant, the simulation will load parametrization from `task_03_swarm:packages/wrapper/config/boids_testing.yaml`. You may use `testing` to play with the task by changing the simulation, the interface, the agents' properties, and many others to discover the potential of your solution. The variants {`easy`, `medium`, `difficult`} will use parametrization specific to the particular variant which will be used during evaluation in BRUTE. Note that particularly useful may be parameter `rate` which allows you to slow down or speed up the simulation.

In BRUTE, your implementation will be **run T times** for each of the {`easy` (T = 100), `medium` (T = 50), `difficult` (T = 50)} variants. The results from the runs will be averaged and used in evaluation (see below). Because of randomized initialization, we allow 10 % failure rate for each of the variants (i.e., 10/100 runs can fail and your solution will still pass). You may run this test locally by calling `./simulation/boids_evaluate.sh`. The test passes if at least 90 % runs pass. The rules for successful completion of a single run are stated in the list at the end of subsection 4.1.

4.6 Scoring

The {`easy`, `medium`, `difficult`} variants are evaluated in BRUTE. Passing the `easy` variant is compulsory, the rest is voluntary. From this task, you may receive from 1 to 16 points as given by Table 1.

Variant (minimum points required)	Alive agents	Base points	Bonus points	Total points
`easy` (1)	$\geq 90\,\%$	1	2	3
	$80\,\% \leq x < 90\,\%$	1	1	2
	$70\,\% \leq x < 80\,\%$	1	0	1
	$< 70\,\%$	0	0	0
`medium` (0)	$\geq 95\,\%$	2	3	5
	$90\,\% \leq x < 95\,\%$	2	2	4
	$80\,\% \leq x < 90\,\%$	2	1	3
	$70\,\% \leq x < 80\,\%$	2	0	2
	$< 70\,\%$	0	0	0
`difficult` (0)	$\geq 95\,\%$	3	5	8
	$90\,\% \leq x < 95\,\%$	3	4	7
	$85\,\% \leq x < 90\,\%$	3	3	6
	$80\,\% \leq x < 85\,\%$	3	2	5
	$75\,\% \leq x < 80\,\%$	3	1	4
	$70\,\% \leq x < 75\,\%$	3	0	3
	$< 70\,\%$	0	0	0
	Max:	6	10	16

Table 1: Scoring of task **T.1**. The ratio of alive agents represents the alive-to-all ratio averaged from 90 % best (most alive agents) runs for each variant. Obtaining at least 1 point from the `easy` variant is compulsory to pass the test.

5 Task T.2: Hunting the robot

5.1 Objectives

The task is similar to the hunt-the-robot task in `task_02_formation` — a source of signal is moving through a structured environment and the primary objective is to find and follow it with a set of UAVs. In contrast to centralized formation control in `task_02_formation`, the control in this assignment is **decentralized**. The goal is to write a control approach which will be run N times in N independent asynchronous instances with **identical parametrization**.

The task has the following properties.

- The number of UAVs is fixed to N = 3.
- Movement of the agents is limited to 2D with all the UAVs flying in the same height.
- Velocity of the UAVs is saturated at $0.75\,\mathrm{m\,s^{-1}}$ (maximal allowed speed).
- Heading of the agents is static and set to zero by default.
- The target (the ground robot) is slowly moving through the free space of the inner 9×9 cells in the environment. The movement is randomized.

The task consists of the following three subproblems.

- Designing and implementing a non-increasing weighting function for UAV-UAV and UAV-obstacle separation.
- Designing and implementing a control action to be passed to the UAVs as velocity reference.
- Implementing a mission logic that will make the decentralized formation of UAVs follow the source of the radio signal.

5.2 Environment

In this task, the UAVs will move in a structured 3D environment depicted in Figure 5. The environment has the following attributes.

- The ground plane is flat and is given as $z = 0\,\mathrm{m}$.
- The ceiling plane is flat and is given as $z = 8\,\mathrm{m}$.
- The collision-free space is modeled as a grid of cylinders with radius of 5 m. The grid contains 11×11 cells with their centers spaced 10 m apart.
- 2 m wide corridors are cut along the grid lines.

The UAV formation will start at the $[0, 0]$ cell; however, the positions of the individual UAVs will be randomized.

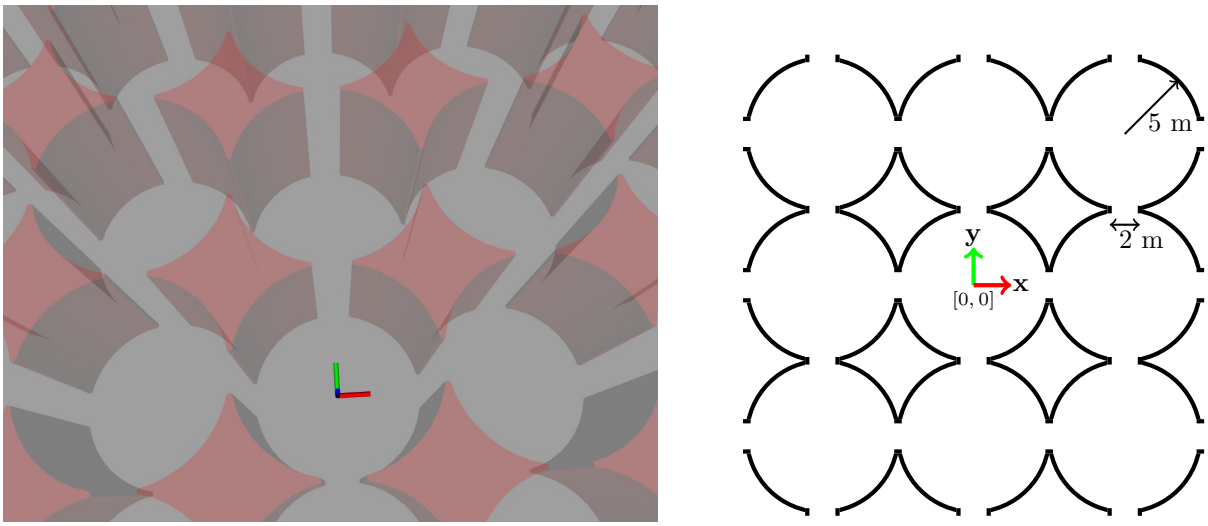

Figure 5: Depiction of the simulated environment in RViz (left), schematic of the top-down ortho view of free space within the simulation environment (right).

5.3 Perceptual inputs

The control of a UAV $\mathcal{A}$ have the following perceptual inputs:

- Current time.
- A unit-vector direction towards the target in the XY plane (the z-axis component is zero). Beware that this vector is subject to the following inaccuracies.

  - Minority of the agents measure the direction incorrectly.
  - Which agents comprise the erroneous minority is set randomly each time all the agents stop.
  - Each time the minority is changed, the statistical properties of the error change randomly.
- Information about other UAVs within the local spherical neighborhood. For each perceived UAV, the input will contain the following.
  - A vector representing the relative position of the agent.
  - Three variables (two integers, one double) shared from the agent through a simulated asynchronous low-bandwidth communication network.
- Information about the obstacles within the current cell (the absolute coordinates of the cell are not given) as shown in Figure 6.
  - A vector representing the relative position of the closest obstacle in the environment.
  - Four pairs of vectors representing the relative position of the gates in the current cell.

All the relative information is relative to the body frame of $\mathcal{A}$. **Beware** that because only local information is available (except of the target vector), the UAVs have no notion about its global position in the world and hence cannot derive directly the cell they are located in.

Remark 3. *The target vector* $\boldsymbol{t}$ *can be characterized by a single angle*

$$\theta = atan2\left(t_y, t_x\right), \tag{10}$$

and vice versa

$$\boldsymbol{t} = [t_x, t_y, 0]^T = [\cos(\theta), \sin(\theta), 0]^T. \tag{11}$$

The prepared implementation uses concept of directions (see Figure 6). For each cell in the environment, these directions {`RIGHT`, `UP`, `LEFT`, `DOWN`} represent the direction in {positive x-axis, positive y-axis, negative x-axis, negative y-axis} from origin of the particular cell. The orientation of the cell-origin axes matches the world coordinate system. You may or you may not use this prepared concept.

5.4 Implementation

All the *logic* is supposed to be implemented within the `task_03_swarm:packages/swarm/src/swarm.cpp` file by filling these two functions — `weightingFunction()` and `updateAction()`.

The function `weightingFunction()`

```
/**
 * @brief Non-linear weighting of forces.
 *
 * The function is to be non-increasing, non-negative, and grows to infinity as the distance is approaching the lower bound (the safety distance). Below the lower bound (including), the function is to be undefined. Over the visibility range, the function shall return 0.
 *
 * @param distance to an agent/obstacle
 * @param visibility visibility range of the UAVs
 * @param safety distance: min distance to other UAVs or obstacles
 * @param desired distance: desired distance to other UAVs or obstacles (does not need to be used)
 *
 * @return
 *   bool:   True if function is defined for the given distance, False otherwise
 *   double: Weight for an agent/obstacle at given distance, if function is defined for the given distance.
 */
std::tuple<bool, double> weightingFunction(const double distance, const double visibility, const double safety_distance, const double desired_distance);
```

relates to $\omega_s(\mathbf{p}, \mathcal{N})$ defined in (4). It takes distance between two agents and outputs a nonlinear weight for a separation function. This function will be evaluated separately in its own test checking whether the function behaves as expected. Passing this test is compulsory! Using this function in other part of your code is recommended but not obligatory.

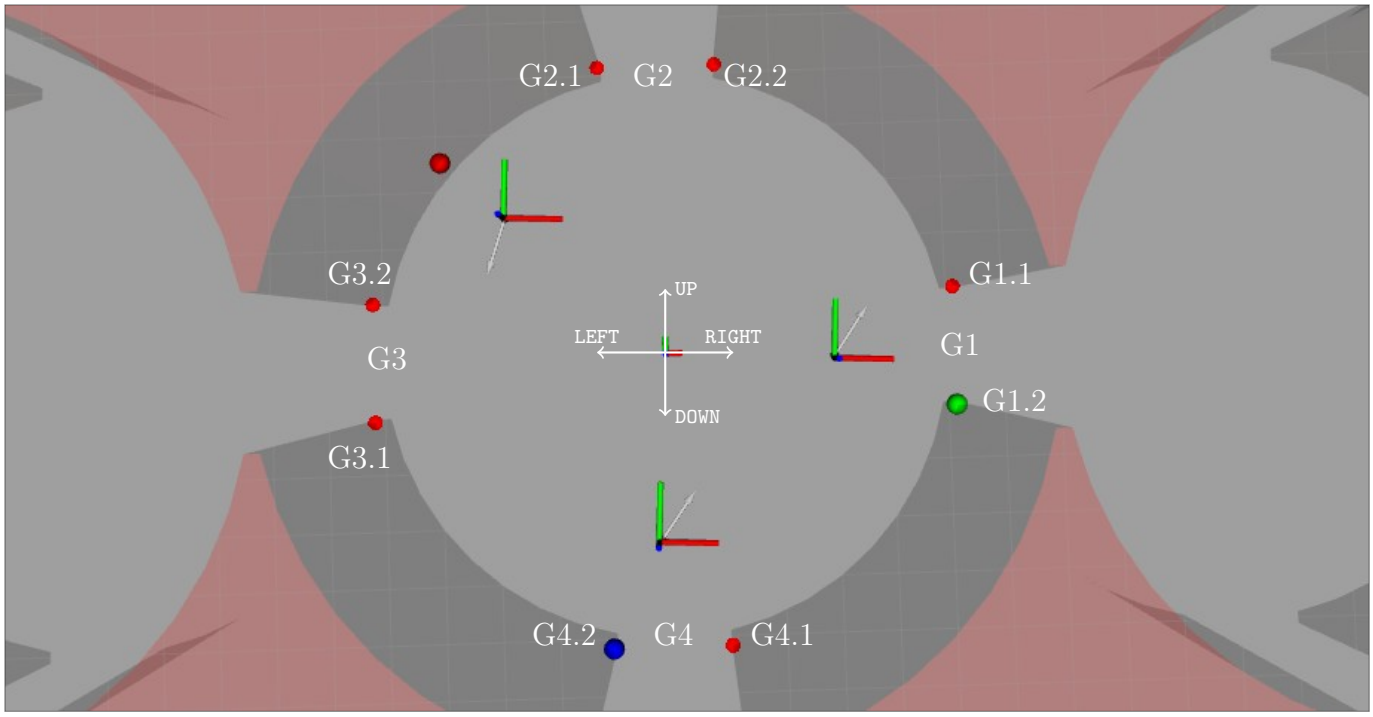

Figure 6: Detail of the default RViz visualization showing three UAVs (the larger axes) in task **T.2**. The smaller axes (partially overlaid with the arrows denoting navigation directions) represent the world coordinate system. The larger RGB spheres show the **closest** obstacle to each of the UAVs. The smaller red spheres represent the edge pair of each gate as detected by one of the UAVs. The gates input to the task are ordered sequentially as denoted, with the Gx.1 representing the first and Gx.2 the second element in each pair G.

Given a distance d (m) between two agents, a visibility distance v (m) of an agent, and a weighting function $f()$ with a lower bound s (m) such as $s < v$, the evaluation will look whether the following criteria are met:

- function is undefined in interval $(-\infty, s\rangle$,
- $f(b) \leq f(a)$ for all $b > a$, where $a, b \in (s, \infty)$,
- $f(d) \geq 0, d \in (s, \infty)$,
- $\lim_{d \to s^+} f(d) = \infty$, and
- $f(d) = 0, d \in (v, \infty)$.

Example of a function passing all the criteria is shown in Figure 1.

The function `updateAction()`

```
/**
 * @brief This method calculates a next-iteration action of one UAV given relative information of ↩
    its neighbors and obstacles; and the direction towards the
 *        moving target. This method is supposed to be filled in by the student.
 *
 * @param perception Current perceptual information of this UAV. Defined in perception.h. It ↩
    contains:
 *  - current time
 *  - target vector: 3D vector towards the moving robot in the UAV body frame
 *  - neighbors defined by:
 *    - their position in the UAV body frame
 *    - the variables shared through the communication network
 *  - obstacles consisting of:
 *    - 3D vector from the body frame to the closest obstacle in the environment
 *    - 4 gates (pairs of 2 gate edges) in the UAV body frame
 * @param user_params user-controllable parameters
 * @param action_handlers functions for visualization and data sharing among the UAVs:
 *  - shareVariables(int, int, double) will share the three basic-type variables among the UAVs
 *  - visualizeArrow() will publish the given arrow in the UAV body frame within the visualization
```

```
 *  - visualizeArrowFrom() will publish the given arrow at position given in the UAV body frame ↩
     within the visualization
 *  - visualizeCube() will publish a cube in the UAV body frame within the visualization
 *
 * @return Next-iteration action for this UAV given as a 3D vector. Zero vector is expected if no ↩
     action should be performed. Beware that i) the vector z-axis component will be set to 0, ii) ↩
     the vector vector magnitude will be clamped into <0, v_max> limits, and iii) the vector will ↩
     be passed to a velocity controller of the UAV.
 *
 *       Example 1: v_max=0.75 -> vector (1, 0, 1) will be saturated to 0.75*(1, 0, 0).
 */
Eigen::Vector3d updateAction(const Perception_t &perception, const UserParams_t &user_params, ↩
    const ActionHandlers_t &action_handlers);
```

is going to be utilized in simulation tests. Remark that the output vector of the `updateAction()` function will be 1) projected to the XY plane, 2) saturated at the desired UAV velocity, and 3) passed as velocity reference to the UAV controller by the task handler calling the function. In the simulation tests, the function will be called at the rate of 10 Hz since the moment all the UAVs are ready in the air. The function is pre-filled to assist you with tackling the task.

The implementation process can be divided into the following fundamental steps:

- Implement the `weightingFunction()` function.
- Design and implement a state machine handling the distributed cooperation, decision making, and navigation through the environment.
- Design and implement a set of local forces to be summed for achieving the desired behavior.
- Tune the weights of your forces to obtain qualitatively smooth behavior.

Remark 4. *The optimal derivation of the weights to achieve equilibria in all scenarios of the task is out of scope of this assignment (and honestly, even out of scope of current state of the art). Design the weights by hand till you achieve qualitatively sufficient collision-free results.*

Remark 5. *Use the functions provided in the `action_handlers` a lot. Particularly the `shareVariables()` without which the task is very hard to be solved.*

5.5 Tips

Designing solution for the task might be unclear, thus you may use the tips below.

- Prepare a state machine logic first. We've prepared some template for you to do that.
  - Note that in some cases, you might want to output zero velocity command.
- Use the `shareVariables()` function to share data between the agents.
- Because of the constraint on proximity of all the agents, you have to synchronize the navigation of all the agents. You may use voting or majority selection in order to do so.
  - In voting, think about what happens if you vote for an erroneous measurement.
  - In majority, think about what happens if all the agents report different value.
- Apart from synchronizing the direction of the agents, you might want to synchronize an order of the agents' movement.
- Use the gates to align the velocity for smoother navigation.
- In various stages of the hunt, you may enable/disable some of the rules/forces you utilize. No need to use them all the time.
- There are various helper functions available either in `swarm.cpp` itself or in `task_03_common/utilities.h`. Using them should save you some implementation time.

5.6 Testing, tuning, and validation

Implementation of the `weightingFunction()` can be validated by running automated test `./simulation/weighting_evaluate.sh`. The hunt-the-robot task implementation can be tested locally by running the `./simulation/run_hunt_the_robot.sh` script.

To **tune** your solution, use parameters passed to the `updateAction()` method via the `user_params` parameter. You may change default values of these parameters in `task_03_swarm:packages/swarm/config/user_params_hunt_the_robot.yaml` or update them on the run in the dynamic reconfigure window (do not forget to store your results in `task_03_swarm:packages/swarm/config/user_params_hunt_the_robot.yaml` after).

The `./simulation/run_hunt_the_robot.sh` task passes if

1. No UAV has landed due to a collision with another UAV or an obstacle.
2. All the UAVs stay together during the hunt. The threshold for maximal allowed distance between two UAVs is 10 m.
3. The UAVs find the robot in time limit of 300 s. To find the robot, all the UAVs have to be in proximity of 7 m to the robot for at least 10 s.

5.7 Scoring

Passing both the automated tests `./simulation/weighting_evaluate.sh` and `./simulation/run_hunt_the_robot.sh` is compulsory. Together, they yield 14 points in total.

6 Starting the simulation

1. You should already have Singularity installed from the previous task. If not, then install the Singularity[2] container software on your system. The CTU university computers will have it installed already. If you are a Linux user, follow the instructions provided for your particular Linux distribution. For Ubuntu, use our pre-configured install script in `task_03_swarm/install/install_singularity.sh`. It is possible to set up Singularity for Windows. However, we provide no support for it. Follow the short manual here[3] for Windows installation instructions.
2. To save your HDD space, you can make a symbolic link of the image from the previous task, e.g., as:

   ```
   ln -s $HOME/task_01_controller/simulation/images/mrs_uav_system.sif $HOME/task_03_swarm/↩
       simulation/images/mrs_uav_system.sif
   ```

   Otherwise, download the pre-built Singularity image by executing: `./simulation/download.sh`. This will download approx. 4 GB of data. The resulting image will be placed in `./simulation/images`.
3. Compile the sources by running `./simulation/compile.sh`.
4. Running task **T.1**:
   - Start the simulation by running `./simulation/run_boids_testing.sh VARIANT`, where `VARIANT` is one of {`testing`, `easy`, `medium`, `difficult`}. The evaluation of the task will start right away in the `task` window.
     - After the simulation starts, three windows will appear on your screen (see Fig. 7):
       (a) A window of the dynamic reconfigure (can be closed if you do not intend to tune the configurable parameters).
       (b) Terminal window running all the simulation software. Feel free to interrupt this windows by *ctrl+c* and restart it again with *up+Enter*. This will restart the entire task, including the evaluation. Nothing else needs to be restarted to reset the task.
       (c) A window of the RViz visualizer.
5. Running task **T.2**:
   - Start the simulation by running `./simulation/run_hunt_the_robot.sh`. The evaluation of the hunt-the-robot task will start automatically after the UAVs take off.

[2] `https://sylabs.io/singularity/`
[3] `https://github.com/ctu-mrs/mrs_singularity/`

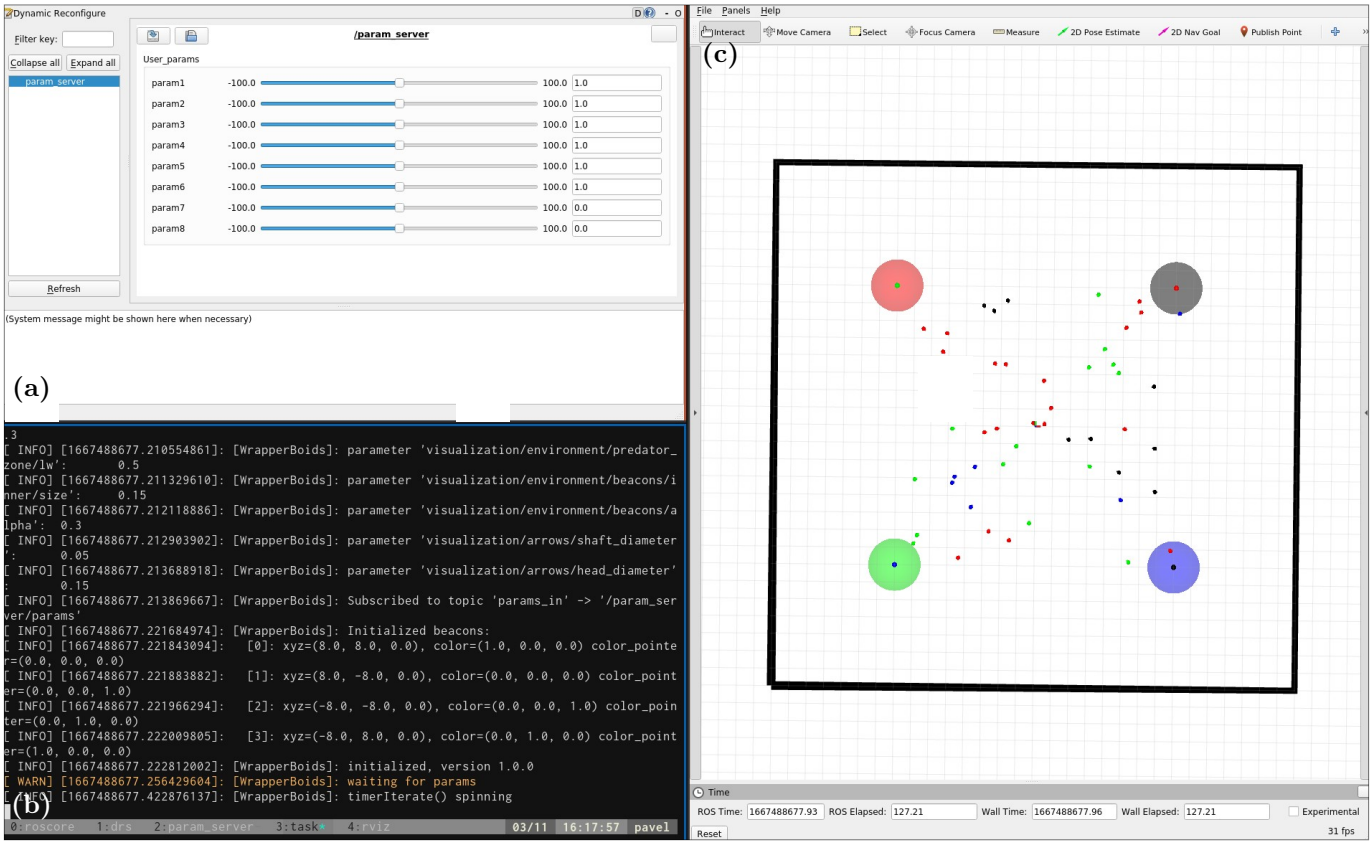

Figure 7: Windows shown during the simulation of task **T.1**: (a) the dynamic reconfigure window allows to change the parameters during the flight (can be closed without affecting the simulation), (b) the terminal window runs your code together with its evaluation, and (c) the RViz GUI shows visualization of task **T.1**.

   - After the simulation starts, four windows will appear on your screen (see Fig. 8):
     (a) Terminal windows running all the simulation software. Three windows (top left, top right, bottom left) run three instances of the student's code. The bottom right window runs evaluation of the hunt task. Feel free to interrupt these windows by *ctrl+c* and restart them again with *up+Enter*.
     (b) A window of the Gazebo simulator (can be closed without affecting the simulation).
     (c) A window of the dynamic reconfigure (can be closed if you do not intend to tune the configurable parameters).
     (d) A window of the RViz visualizer.
6. Kill the simulation by running `./simulation/kill_simulation.sh`.
7. Start a code editor from within the container:
   - start the VSCode editor `./simulation/vscode.sh` or
   - start the Sublime Text editor `./simulation/sublimetext.sh`.

7 Dos and Don'ts

7.1 Dos

- Think about how the task needs to be tackled if 1) you do not have all the information centrally and 2) the information you have is local only. Think about
  - how to write the code just for one homogeneous agent and
  - how to design the data flow and procedures required to coherently decide on the actions of all the agents.

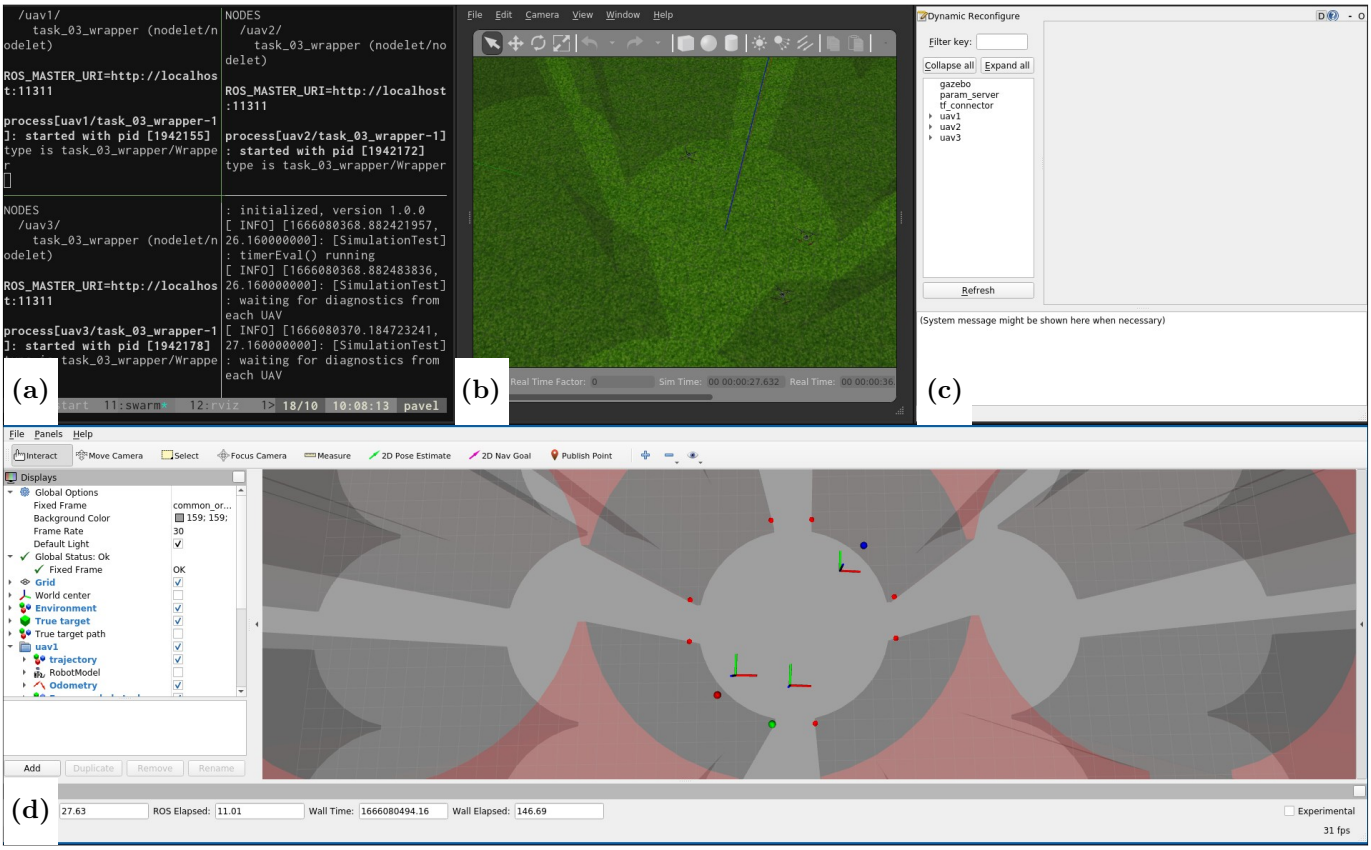

Figure 8: Windows shown during the simulation of task **T.2**: (a) the terminal window runs the student's code in three distributed instances and evaluation of the task in the bottom right corner, (b) the Gazebo GUI shows what exactly are the UAVs doing (**can be closed to increase runtime performance)**, (c) the dynamic reconfigure window allows to change the parameters during the flight (can be closed), and (d) the RViz GUI shows visualization of task **T.2**.

- Study this document and orient yourself within the provided source codes.
- Use the following functions a lot. It will save your time.
  1. `action_handlers->shareVariables(...)`
  2. `action_handlers->visualizeArrow(...)`
  3. `action_handlers->visualizeArrowFrom(...)`
  4. `action_handlers->visualizeCube(...)`

7.2 Don'ts

- Do **NOT** modify any other files except the following:
  - `task_03_swarm:packages/swarm/src/boids.cpp`
  - `task_03_swarm:packages/swarm/src/swarm.cpp`
  - `task_03_swarm:packages/swarm/include/student_headers/swarm.h`
  - `task_03_swarm:packages/swarm/config/user_params_boids.yaml`
  - `task_03_swarm:packages/swarm/config/user_params_hunt_the_robot.yaml`
- Do **NOT** write any data into system files. The automatic evaluation checks for this and will let us know if you would try to do so.

- Do **NOT** use any third-party library, besides Eigen[4], which is already provided.
- Do **NOT** advertise nor subscribe to any ROS topics. The automatic evaluation checks for this and will let us know if you would try to share any information outside the provided communication channel.

8 Submit to BRUTE

The BRUTE server will run the same tests that are available on your local machine. If the tests pass on your local machine, it is likely they will also pass on the server.

1. Upload an archive (zip, tar, tag.gz) of the **swarm** (located in the `packages` folder) folder to BRUTE. Create the archive, e.g., by issuing the command:

```
zip -r swarm.zip swarm
```

2. Upload the archive to BRUTE (`https://cw.felk.cvut.cz/brute/`).
3. Wait for the evaluation.
4. If the compilation fails, you will be presented with the compilation log.
5. If the compilation succeeds, you will be presented with logs from the **T.1 evaluation**, and the **T.2 weighting test** and **simulation**.

The evaluation can take up to 15 minutes.

[4] `https://eigen.tuxfamily.org/dox/GettingStarted.html`