B(E)3M33MRS - Task 02 Formation

Tomáš Báča

November 8, 2023

v1.01

The second task focuses on centralized formation control and planning for multirotor Unmanned Aerial Vehicles (UAVs).

1 Requirements

- Implement the method for planning 3D collision-free paths for reshaping a UAV formation.
  - update the `getPathsReshapeFormation()` method in `task_02_formation:packages/formation/src/formation.cpp`
- Finish the provided `multilateration()` method. You can further improve it to estimate the position of the signal source more reliably.
- Implement a multi-robot control strategy for localization and following of the source of signal.
  - program the contents of the `update()` function in `task_02_formation:packages/formation/src/formation.cpp`

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

2 Preliminaries

Formation control is a multi-robot control scheme that can be implemented with a varying level of centralization. This task will focus on exploring aspects of sensor-driven centralized formation control. An aerial formation, consisting of several identical multirotor UAVs, will be tasked with search for a source of radio signal. The source of radio signal is moving through an environment which is densely-populated with obstacles. We consider each UAV already equipped with a well-tuned feedback and feedforward dynamics controller. Each UAV is also capable of following a provided 3D path consisting of a series of 3D waypoints.

2.1 Virtual-leader formation control

Maintaining a compact formation can sometimes be the only way of achieving desired results, e.g., due to a limited effective communication range of the agents. To maintain compactness of a formation, we will use the concept of *virtual leader* serving as a central element for all the agents.

The state of a formation consisting of N UAVs is described by

- $\mathbf{x}_{\mathrm{VL}} \in \mathbb{R}^3$: position of the *virtual leader* (frame $\mathcal{L}$) in world frame $\mathcal{W}$,
- $\mathbf{x}_k \in \mathbb{R}^3, k \in 1, \ldots, N$: position of the k-th UAV expressed in $\mathcal{W}$,
- $\mathbf{x}_k^{\mathcal{L}} = \mathbf{x}_k - \mathbf{x}_{\mathrm{VL}}, k \in 1, \ldots, N$: position of the k-th UAV expressed in $\mathcal{L}$.

The position of the virtual leader can be arbitrary in general, it does not have to coincide with any of the real UAV positions. The *virtual leader* is used as an *anchor* in the formation, through which the formation motion can be controlled. In our case, the formation **can be moved as a whole** by setting a desired position to the *virtual leader*.

On the other hand, **changing the shape** of the formation is done by commanding each UAV to move relatively to the *virtual leader's* position.

2.2 Multi-robot collision-free planning

The term *planning* in this task is going to be used specifically for the operation of searching a collision-free *path* through a 3D environment from position A to a position B. A path is a sequence of *waypoints* (3D positions), that can be connected by straight collision-free segments. The path planning does not take the UAV dynamics into account, and therefore does not need to work with time parametrization. When a path is issued to the UAV, it will fly along the segments, until it stops at the last supplied waypoint. Given a set of dynamics constraints, the flight is governed by the feedforward controller. The student can not influence when the UAV is going to be at which part of the path. **Therefore, paths for our formation of UAVs need to be completely collision-free in space to guarantee no collisions will happen anytime during the flight.** Searching for collision-free paths can be done in many ways. For small formations in an obstacle-free environment, a specific solution can be tailored, e.g., separation of the UAVs to different flight altitudes (don't forget about special cases). For large formations or complex environments, a planning algorithms need to be involved, such as

- Graph-based: **A***, Dijkstra's,
- Sampling-based: RRT, RRT*.

Existing **implementation of A*** including the detailed description for its usage is provided in `task_02_formation: packages/formation/include/students_header/astar.h`.

2.3 Multilateration

Multilateration is a technique used to estimate the position of an object based on measured distances to a set of landmarks. A typical use case for multilateration is a Global Navigation Satellite System (GNSS) localization system, such as the Global Positioning System (GPS). In GPS, the position of a personal receiver is computed based on the measured distances between the receiver and a set of Earth-orbiting satellites given the satellites' known positions. Similarly, in the case of this task, multilateration will be used to calculate the position of a **moving ground robot**, with the UAVs serving as the "satellites". The mobile robot is equipped with a simulated Ultra-Wide Band (UWB) radio-beacon, and each of the UAVs within the formation is equipped with an appropriate signal receiver. Therefore, each UAV is capable of measuring the approximate distance between itself and the mobile robot.

In ideal world, the position of the target would be found as intersection of spheres, each originating from a UAV, having the radius of the measured distance. For that, the following set of equations would hold:

$$\|\mathbf{s} - \mathbf{x}_k\| = d_k, k \in 1, \ldots, N, \tag{1}$$

where $\mathbf{s} \in \mathbb{R}^3$ is the position of the signal source (the ground robot), $\mathbf{x}_k \in \mathbb{R}^3$ is the position of k-th UAV and d_k is the measured distance from the k-th UAV to the signal source. However, in the real world, the intersection does not exist in general, due to noise in the distance measurements, as well as inaccuracy of the UAV localization. For that reason, the equation (1) does not have a solution. One possible approach to find good estimate of $\mathbf{s}$ is to minimize the sum of squares of residuals of the equations in (1):

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \sum_{k=1}^{N} \left(\|\mathbf{s} - \mathbf{x}_k\| - d_k\right)^2. \tag{2}$$

This continuous optimization problem can be tackled in many ways. A common way is to exploit the particular form of the criterion function which leads to the *nonlinear least square method*, in which we minimize a function $f(\mathbf{x}) = \|\mathbf{g}(\mathbf{x})\|^2 = \mathbf{g}(\mathbf{x})^\intercal \mathbf{g}(\mathbf{x}) = \sum_{i=1}^{M} g_i(\mathbf{x})^2$, where g_i are the elements of the multi-variate function $\mathbf{g}$. In our case

$$g_k = \|\mathbf{s} - \mathbf{x}_k\| - d_k, k \in 1, \ldots, N. \tag{3}$$

Such *nonlinear least squares* problem can be solved, e.g., iteratively by the Gauss-Newton method. The Gauss-Newton step will take the form

$$\mathbf{s}_{n+1} = \mathbf{s}_n - \left(\left(\mathbf{J}_\mathbf{g}(\mathbf{s}_n)^\intercal \mathbf{J}_\mathbf{g}(\mathbf{s}_n)\right)^{-1} \mathbf{J}_\mathbf{g}(\mathbf{s}_n)^\intercal\right) \mathbf{g}(\mathbf{s}_n), \tag{4}$$

where $\mathbf{J_g}$ is the Jacobi matrix of the map $\mathbf{g}$ given as

$$\mathbf{J_g}(\mathbf{s}) = \begin{bmatrix} (\mathbf{s}-\mathbf{x}_1)^\intercal \,/\, \|\mathbf{s}-\mathbf{x}_1\| \\ (\mathbf{s}-\mathbf{x}_2)^\intercal \,/\, \|\mathbf{s}-\mathbf{x}_2\| \\ \vdots \\ (\mathbf{s}-\mathbf{x}_N)^\intercal \,/\, \|\mathbf{s}-\mathbf{x}_N\| \end{bmatrix} \in \mathbb{R}^{N\times 3}. \tag{5}$$

However, just solving the optimization problem using Eq. (4) does not necessarily yield good results. The objective of this homework assignment is to improve the already implemented Gauss-Newton optimization, in order to obtain more reliable results, by, e.g.,

- varying or randomizing the initial conditions for the source hypothesis $\mathbf{s}_0$,
- initializing $\mathbf{s}$ using previous good estimate of $\mathbf{s}^*$,
- rejecting outlier solutions,
- averaging the results (or measurements) over longer period of time,
- using, e.g., Kalman Filter to further filter the robot's coordinates, or
- modifying the problem (2) to better reflect reality, e.g., by also minimizing the robot's distance from the ground plane.

2.4 Starting the simulation

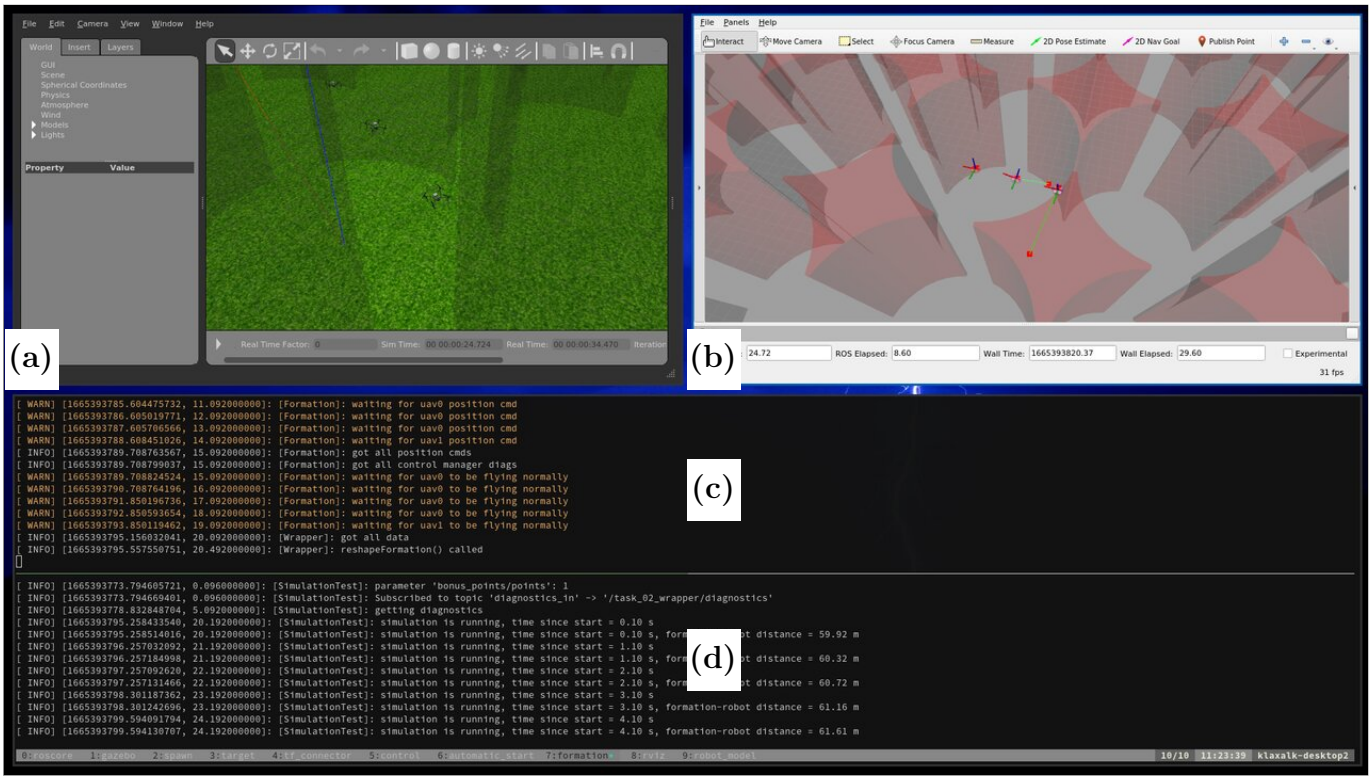


Figure 1: Windows shown during the simulation: a) the Gazebo GUI shows what exactly are the UAVs doing (**can be closed to increase runtime performance**), b) the RViz GUI shows visualization of some internal data, c) the terminal window running the student's code, d) the terminal window running the evaluation program.

1. You should already have Singularity installed from the previous task. If not, then install the Singularity[1] 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_02_formation/install/install_singularity.sh`. It is possible to set up

[1] `https://sylabs.io/singularity/`

Singularity for Windows. However, we provide no support for it. Follow the short manual here[2] 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_02_formation/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 for the controller by running `./simulation/compile.sh`.

4. Start the full simulation by running `./simulation/run_simulation.sh`. The evaluation of the full task will start automatically after the UAVs take off.

   - After the simulation starts, four windows will appear on your screen (see Fig. 1):

     (a) A window of the Gazebo simulator.

     (b) A window of the RViz visualizer.

   (c), (d) Terminal windows running all the simulation software. The window on top contains the student's code. Feel free to interrupt it by *ctrl+c* and restart it again with *up+Enter*.

5. The evaluation of the *hunt for the robot* will start automatically after take off.

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`.

3 Environment

In this task, the UAVs will move in a structured 3D environment, depicted in Figure 2. In order to make the task as simple as possible, we designed the environment to be simple to parametrize and to model. The environment has the following attributes.

- The ground plane is flat and is given as $z = 0\,\text{m}$.
- The ceiling plane is flat and is given as $z = 8\,\text{m}$.
- The collision-free space is modeled as a grid of cylinders with radius of 5 m. The grid contains 19×19 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.

A source of radio signal (the ground robot) is moving randomly through the free space of the environment. The distances to the robot are measured by each of the UAVs individually and these measurements are processed centrally by the `multilateration()` method.

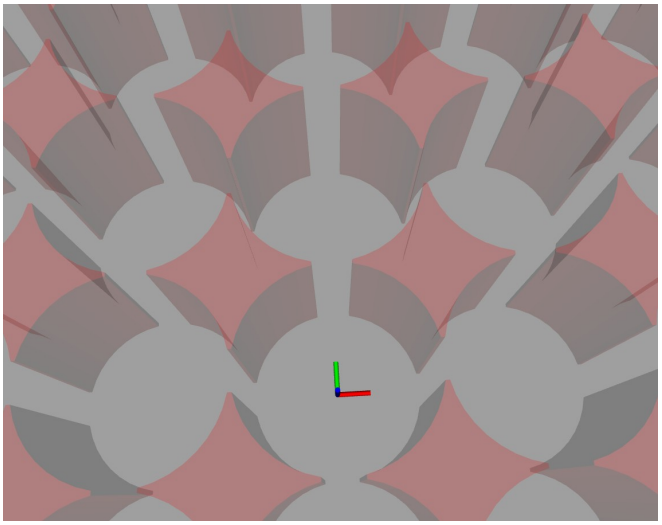
Remark 1. *It is useful to move the formation from one cylindrical cell to another. Each cell center is at* $\left(a \begin{bmatrix}10\\0\end{bmatrix} + b \begin{bmatrix}0\\10\end{bmatrix}\right)$, *where* $a, b \in \{-9, -8, \ldots 8, 9\}$.

4 Tasks

The tasks consist of the following three subproblems:

- Creating a formation-reshaping planning algorithm.

[2] `https://github.com/ctu-mrs/mrs_singularity/`



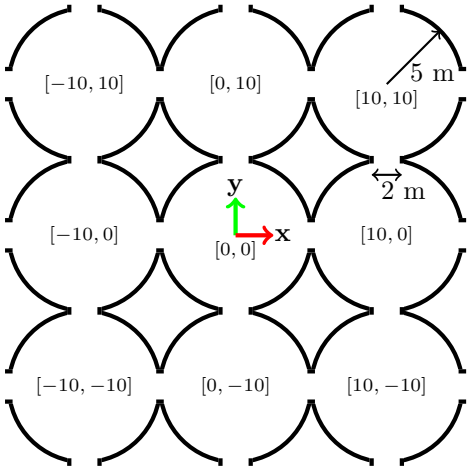


Figure 2: Depiction of the simulated environment in RViz (left), schematic of the top-down ortho view of free space within the simulation environment (right).

- Improving the provided multilateration method.
- Implementing a mission logic that will make the formation of UAVs follow the source of the radio signal.

4.1 Formation reshaping

Reshaping a formation of UAVs is crucial for fulfilling this task. The formation needs to be reshaped to:

- safely move through the environment,
- obtain valid measurements of the target's position.

To reshape a formation, students are supposed to plan paths for each UAV such that each UAV moves from its pre-defined initial position (the current UAV position) to a desired final position. The paths need to be free of any collisions. **The threshold for mutual collision is 1.2 m distance between the centers of two UAVs**.

The final positions are not strictly defined for each UAV, since all the UAVs are functionally equivalent. Instead, the list of final positions **can be re-ordered** to make the reshaping as fast and as safe as possible. Therefore, e.g., if a formation of two UAVs is tasked to swap their positions, the final positions can be reordered, such that each UAV stays at its original position. Pairing of the initial positions and the final positions is a special case of the **minimum cost bipartite graph matching** problem.

> **Remark 2.** *When using the grid-based planners, the found path is constrained by the planning grid. Thus, although some grid-based algorithms such as A* or Dijkstra are known to provide an optimal solution, the produced path can be significantly longer than the optimal solution uninfluenced by the constraints imposed by the planning grid. This limitation can be overcome by post-planning path straightening or by using any angle variants of grid-based planners, e.g., Theta*.*

4.1.1 Implementation of formation reshaping

The formation reshaping algorithm is supposed to be implemented into the prepared `getPathsReshapeFormation()` function, within `task_02_formation:packages/formation/src/formation.cpp`.

```
std::vector<std::vector<Eigen::Vector3d>> getPathsReshapeFormation(const std::vector<Eigen::↩
    Vector3d> &initial_states, const std::vector<Eigen::Vector3d> &final_states);
```

The function receives two inputs: the vector of initial positions of the UAVs and the vector of final positions of the UAVs. The output of the function is a vector of paths for each UAV, each path being a vector of positions along the path.

The formation reshaping method will be evaluated in isolation automatically in the BRUTE system. Please, make sure you follow the following method requirements.

- **Each UAV path must start at the initial state and must end at one of the final states.** Therefore, the most naive solution (already implemented, not acceptable) is to return the straight path from the initial to the corresponding final state. A more elaborate solution is required in order to avoid mutual collisions between the UAVs.
- The threshold for mutual collision of two paths is **1.2 meter, measured between the centers of the robots**.
- Obstacles, ground floor and ceiling are **not** considered while evaluating the method in BRUTE.
- The method will be tested for 2–10 UAVs in BRUTE. You need to pass the test for up to 5 UAVs to get a pass for this subtask. Success with more UAVs will earn you bonus points (1 point for each additional UAV).

For the simulation, the mutual collision constraints are not enforced. Students are supposed to call `getPathsReshapeFormation()` on their own to obtain desired paths. Students will pass the resulting paths to `action_handlers->reshapeFormation(...)`, which only checks whether:

- the initial point of the path reflects the state of the formation,
- distance of each point of each path is $\leq 9\,\mathrm{m}$ from the virtual leader.

Students are advised to use the provided implementation of the **A*** algorithm, located in `task_02_formation: packages/formation/include/student_headers/astar.h`. The implementation provides a simple and intuitive interface to the planner. Please, refer to the header file for instructions.

> **Remark 3.** *The provided **A*** algorithm treats the robot as a compact point. Make sure, you are providing a map with obstacles being inflated in order to avoid mutual collision. The threshold for detecting mutual collision between two UAVs is 1.2 meters between the UAV centers.*

4.1.2 Validation, testing and evaluation of formation reshaping

Development and debugging of the multirobot formation reshaping algorithm can be tedious using the full simulation environment. For that reason, we provide a *quick testing program*, that will execute the method on a given problem and visualize the results. Execute the script `task_02_formation:simulation/reshaping_test.sh` to test and visualize the method. The definition of the particular problem is located in `task_02_formation:packages/formation/config/reshaping_debug.yaml`, feel free to change the definition to your needs.

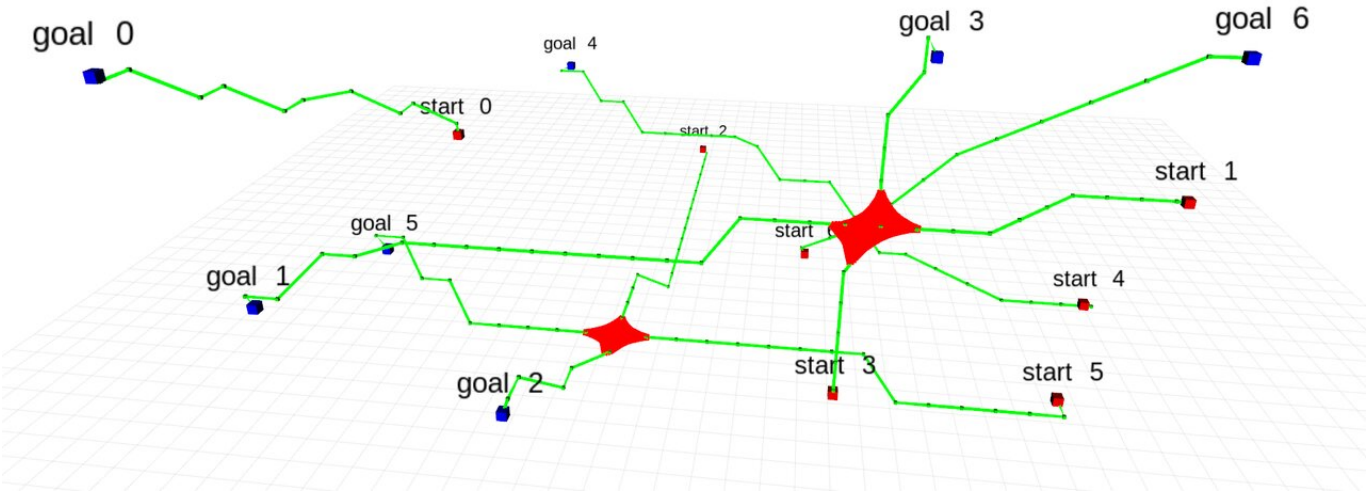


Figure 3: Visualization of results of the reshaping method. Red cubes mark the start nodes, blue cubes mark the goals, green cubes mark the student's paths' waypoints, green segments mark the student's paths and red segments mark the collisions between the paths.

After the method has been developed and debugged, use the script `task_02_formation:simulation/reshaping_evaluate.sh` to evaluate the method on dataset of problems for 2–10 UAVs.

4.2 Multilateration

The provided multilateration is very basic and can be significantly improved for achieving better results. The algorithm is implemented in the `multilateration()` function in the file `task_02_formation:packages/formation/src/formation.cpp`

```
Eigen::Vector3d multilateration(const std::vector<Eigen::Vector3d> &positions, const Eigen::↩
    VectorXd &distances);
```

Even though the **easy** variant of the assignment can be completed with the pre-existing multilateration method, the **difficult** variant requires improving the multilateration method. To achieve the maximum score, follow the tips provided in Section 2.3. This subtask **will NOT be** evaluated in BRUTE individually.

4.2.1 Working with the results of multilateration

The baseline implementation of multilateration provides a single 3D position estimate of the robots position based on the position of the UAVs and their ranging measurements. However, due to the noisy ranging measurements, the estimates can wildly differ between iterations. We recommend to aggregate measurements over a longer period of time (seconds) and to conduct some computations for obtaining better results. The computations may involve on or more of the following:

- outlier rejection,
- rejection of results containing *NaN* or *Inf* values,
- averaging,
- using LKF.

4.3 Hunt for the robot

Finally, the main task for this assignment is to utilize the `multilateration()` and the `getPathsReshapeFormation()` methods to hunt the signal source (the mobile ground robot) with the formation. This is a *creative* task without a strict guideline for approaching it. The goal is to reach the mobile ground robot with the formation as quickly as possible and to follow it closely for at least 1 minute.

4.3.1 Implementation

All the *logic* is supposed to be implemented within the `task_02_formation:packages/formation/src/formation.cpp` file, particularly into the `update()` function:

```
void update(const FormationState_t &formation_state, const Ranging_t &ranging, const double &↩
    time_stamp, ActionHandlers_t &action_handlers);
```

The `update()` function is going to be called at the rate of 10 Hz since the moment the formation is ready in the air. The function's arguments will provide the following:

- `formation_state`: the 3D position of the *virtual leader* (the average of all UAVs' positions), the 3D position of each UAV relative to the *virtual leader*, and a boolean flag stating whether the formation is static (ready to be moved).
- `ranging`: measured distances between each UAV and the ground mobile robot.
- `time_stamp`: current time in *seconds* (the simulation starts at $t = 0$).
- `action_handlers`: object providing methods for visualization and controlling the formation. The following methods are available to the students through the `action_handlers` object in the `update()` method.
  - `action_handlers->setLeaderPosition(...)`: will move the *virtual leader* and the formation with him. The formation will fly along a straight path, no mutual collisions will happen. The function will only move the formation when `formation_state->is_static` is `true`. **The function expects the input in the world reference frame**, i.e. absolute coordinates.
  - `action_handlers->reshapeFormation(...)`: will reshape the formation by following supplied paths for each UAV. Use the `getPathsReshapeFormation()` method to obtain the paths. The function will only

reshape the formation when `formation_state->is_static` is `true`. **The function expects the input in the reference frame of the virtual leader**, i.e. relative to the virtual leader.

- `action_handlers->visualizeCube(...)`: will visualize a cube in the RViz visualizer. The position, color, size and opacity of the cube can be defined. This is a useful way to check for results of your calculations. The cube will stay in RViz until another cube with the same *topic name* is visualized again. The cube's coordinates are interpreted in the absolute world reference frame.

5 Dos and Don'ts

5.1 Dos

- Use the `action_handlers->visualizeCube(...)` a lot, it will save you a lot of time.
- If your computer struggles with the simulation, disable the simulator's GUI in `task_02_formation:simulation/{tmux_simulation_easy,tmux_simulation_difficult}/session.yaml` by replacing `GUI=true` with `GUI=false` on line 3.

5.2 Don'ts

- Do **NOT** modify any other files than the following:
  - `task_02_formation:packages/formation/src/formation.cpp`
  - `task_02_formation:packages/formation/include/student_headers/formation.h`
  - `task_02_formation:packages/formation/include/student_headers/astar.h`
  - `task_02_formation:packages/formation/config/reshaping_debug.yaml`
- Do **NOT** use any third-party library, besides Eigen[3], which is already provided.

6 Evaluation

Students' final score will be based on evaluation of the two subtasks: the performance of the `getPathsReshapeFormation()`, and the whole task of `hunt for the ground robot`. If both the subtasks are fulfilled, the student receives the **base score** of 10 points. Additionally, 15 bonus points can be earned by improving both solutions beyond the minimum requirements.

- The `getPathsReshapeFormation()` function will be evaluated for formations of 2 – 10 UAVs. The minimum accepted solution needs to work for at least 5 UAVs. Students will earn 1 point for each increase in cardinality of the formation, for which the provided solutions works (1 point for 6 UAVs up to 5 points for 10 UAVs).
- The simulation of the `hunt for the ground robot` will be evaluated based on the distance of the virtual leader to the ground robot. The minimum accepted solution needs to reach and follow the ground robot in distance lower than 20 m for at least 1 minute. Two difficulty variants are provided for this subtask:
  - **Easy** (10 points, default): The easy variant can be fulfilled without modifying the baseline multilateration method.
  - **Difficult** (20 points): The difficult variant requires the improve the baseline multilateration in order to succeed. Two aspects of the simulation are different: 1) the ground robot starts it's path further away from the UAV, 2) the ranging measurements are more noisy.

6.1 On your local machine

6.1.1 getPathsReshapeFormation()

The performance of your implementation can be tested by running the `./simulation/reshaping_evaluate.sh` script. When it fails, the problem at which it failed will be printed to the terminal. You can copy the problem definition to `task_02_formation:packages/formation/config/reshaping_debug.yaml` and run the `./simulation/reshaping_test.sh` script to see the visualization for the particular problem.

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

6.1.2 Hunt for the ground robot

Start simulation of the **easy** variant using the script `./simulation/run_simulation_easy.sh`. Start simulation of the **difficult** variant using the script `./simulation/run_simulation_difficult.sh`. The terminal window output (the *formation tab*) will inform you about the task being fulfilled.

6.2 On the BRUTE server

The BRUTE server will run the same two 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. By default, the **easy** variant is evaluated. If you wish to switch BRUTE to evaluation the **difficult** variant, delete the file `task_02_formation:packages/formation/VARIANT_EASY.txt`.
2. Upload an archive (zip, tar, tag.gz) of the **formation** (located in the `packages` folder) folder to BRUTE. Create the archive, e.g., by issuing the command:

```
zip -r formation.zip formation
```

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

The evaluation can take up to 15 minutes.

7 Change log

- 2023-11-08 fixed setLeaderPosition() function name