Aerial and underwater robots

AUV simulation and control

1 Content of this lab

The goal of this lab is to design a thruster-propelled Autonomous Underwater Vehicle (AUV). After the geometric design, a rough approximation of the hydrodynamic damping will be computed and low-level control (PID's) will be tuned. At the end, the AUV should be able to follow a trajectory defined with a sequence of waypoints.

1.1 Installing the simulator

The simulator is in a ROS package called `freefloating_gazebo` and should be already installed on the P-ro computers.
The actual package that you will modify is downloaded (in `~/ros/src`) with `rosdl ecn_rasom`[1]
This package has the classical ROS structure:

```
ecn_rasom
├── config
│   └── waypoints.yaml
├── launch
│   ├── auv.launch
│   ├── auv_estim.launch
│   └── ekf.launch
├── scripts
├── src
│   └── waypoint.cpp
├── subject
├── urdf
│   └── auv.xacro
├── package.xml
└── CMakeLists.txt
```

Figure 1: Files used by the simulator

2 Waypoint following

The first task is to finish the waypoint following node defined in `waypoint.cpp`. This node already loads the waypoints defined in `waypoints.yaml` but does not follow them yet. These waypoints should be modified according to the trajectory you want the robot to follow. A

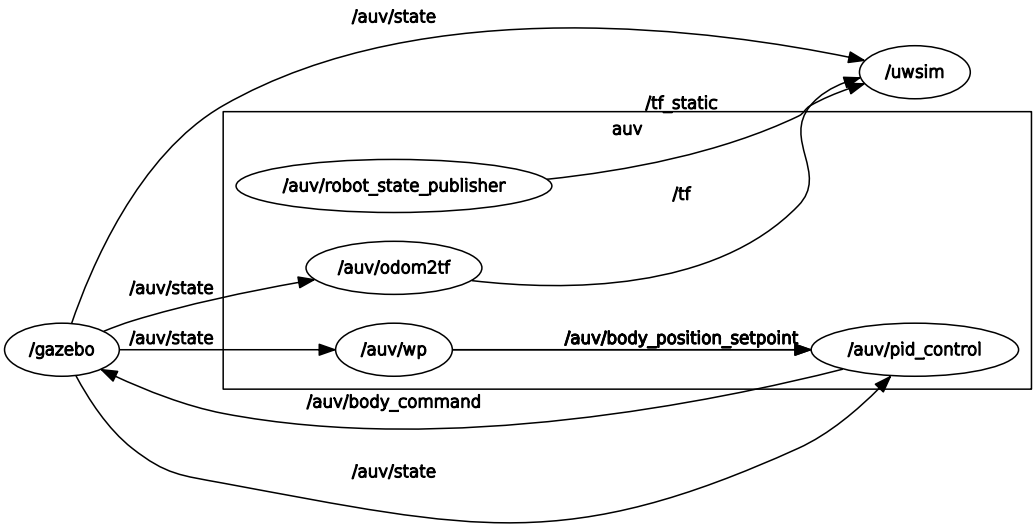
[1] to use your own computer, git should be used to clone https://github.com/oKermorgant/ecn_rasom.git

waypoint is defined by its (x, y, z, θ) values where θ is the yaw angle. Look at the current code to see how to change a yaw angle to a quaternion needed by the `PoseStamped` message.

Topics can be renamed or the node can be run through a launch file in order to be in the `/auv` namespace. The simulation can then be run with:

- `roslaunch ecn_rasom auv.launch` for all built-in nodes
- `rosrun ecn_rasom waypoint` for yours

which leads to the following graph where `/auv/wp` is your node.



Two GUI are launched:

- UWsim is an underwater rendering software that displays the AUV in a realistic environment
- RViz is a visualizer and displays the current pose estimate (red) and the current waypoint to be followed (green)

The Gazebo dynamic simulator is also launched without GUI. In case of issues with UWsim display, the launch file can be used with Gazebo GUI instead of UWSim:

```
1 roslaunch ecn_rasom auv.launch uwsim:=false
```

Manually defined setpoints can be easily published by using `rqt` message publisher, to check if the waypoints can be followed (maximum depth, etc.).

The goal is to modify the source and waypoint list in order to loop through the waypoints (go back to waypoint 0 after the last one was reached). The structure of the `setpoint` message can be found with:

```
1 rosmsg show geometry_msgs/PoseStamped
```

which gives:

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

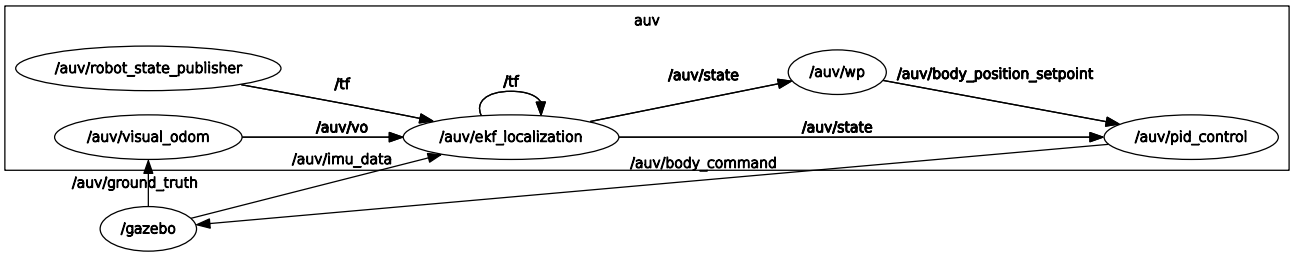
The header is already written but you have to define the pose section from the reading of the loaded waypoint file. A function is available to write the $i-$th waypoint into the `setpoint` message:

```
writeWP(int idx);
```

Remember that the AUV may not reach perfectly the desired position, so you should cycle to the next waypoint when the position error is below a given threshold. Two thresholds are loaded from the YAML file: `thr` and `thr_angle`, that may be tuned accordingly to the desired accuracy.
The current position of the AUV can be found on the topic `/auv/state` and is of type `nav_msgs/Odometry` with a structure similar to `geometry_msgs/PoseStamped`, and where we are interested only in the `pose` section.

3 Pose estimation from Extended Kalman Filter

The architecture used until then is quite unrealistic. Indeed, data on the topic `/auv/state` is the ground truth in terms of absolute position and velocity of the AUV (it is coming directly from the simulator). In this section we will replace it by an EKF estimator and see the problems that arise. The new graph will be:



Here, Gazebo is still publishing the ground truth but on the `/auv/ground_truth` topic. This data is processed (noise is added) and republished from two simulated sensors:

1. A depth sensor publishing a pose on `/auv/depth` where only the Z-position is meaningful

2. A USBL device publishing a pose on `/auv/usbl` where only the X and Y positions and Z orientation (yaw) are meaningful

Gazebo is also publishing simulated IMU data on the `/auv/lsm` and `/auv/mpu` topics. Finally, the Gazebo gui is not visible anymore (like an actual underwater vehicle) and RViz displays the estimate of the AUV (in red). For comparison purpose, the ground truth is displayed in blue. The setpoint is still displayed in green.
The sensor data (depth, USBL and 2 IMU) should be fused by the EKF node in order to output an estimate of the state on the `/auv/state` topic that is used by the PID and your waypoint node.
Two launch files have to be run to do so:

- `roslaunch ecn_rasom auv_estim.launch` for all built-in nodes
- `roslaunch ecn_rasom ekf.launch` for the EKF and your node

The `ekf.launch` should be modified in order to:

- run your waypoint node
- use the EKF node with available sensors

Documentation of the EKF node can be found on the `robot_localization` package website. This node creates a 15-components (6 pose, 6 velocities and 3 accelerations) state estimate. The parameters of this node should be modified so that the EKF node subscribes to the available data topics. The boolean tables should also be updated in order to tell which components of the state are measured by each of the sensors.
Do not hesitate to run the EKF with only a partial knowledge of the state (only IMU for instance) in order to see the effect on the localization accuracy.