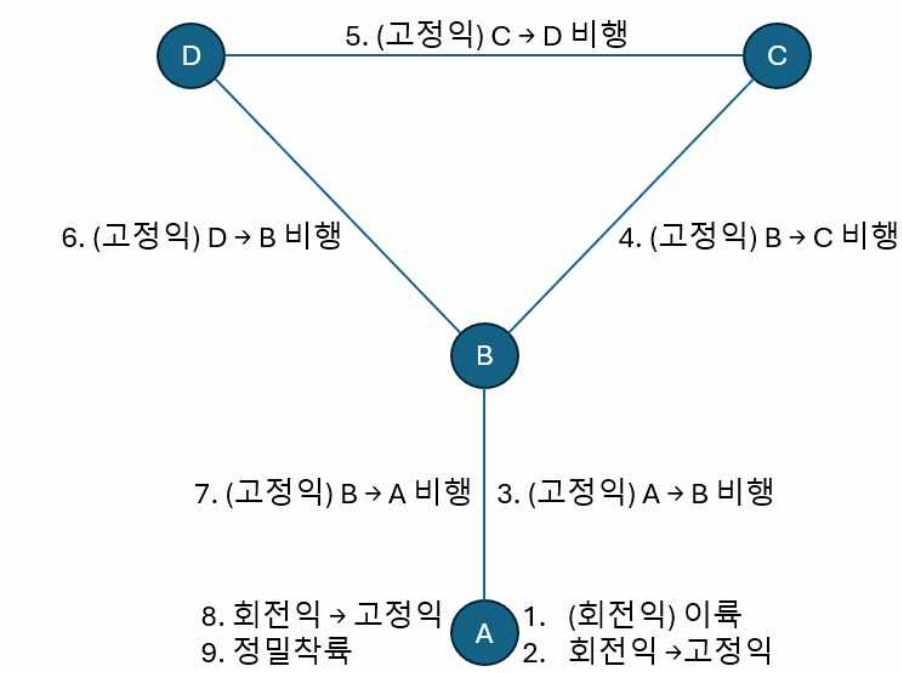
연세대학교
Space-Y 항공팀 제어부
2025/11

VTOL 자율비행 및 정밀착륙 시뮬레이션

1. 개요

비행체 비행 및 임무 완수를 위한 프로그램 개발 후 검증은 위해 시뮬레이션을 통해 이루어진다. 시뮬레이션을 활용하면 비행체 제작과 제어 프로그램 개발이 동시에 가능하고, 제어 프로그램의 탑재 전 검증을 통한 안정성 확보가 가능하다. 해당 문서에서는 VTOL 기체의 자율비행 및 ArUco 마커를 활용한 착륙 프로그램을 시뮬레이션에서 실행하는 방법을 다룬다. 실행 환경은 “항공팀 개발환경 통일안 v1.0”을 기준으로 하며, 구축 방법은 “개발환경 구축 방법 v1.1”을 참고하면 된다.



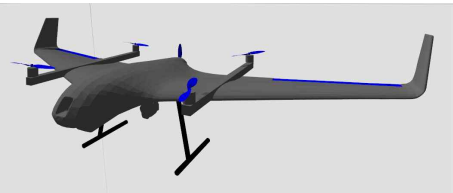
<그림 1>

위의 그림은 시뮬레이션에서 기체가 수행하게 될 임무이다. 출발 지점에서 회전익으로 이륙 후 고정익으로 전환하여 참고점을 지나 다시 회전익으로 전환, ArUco 마커를 통한 정밀 착륙을 진행한다.

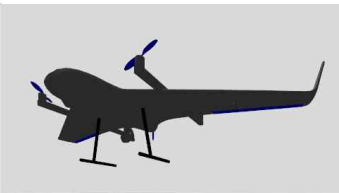
10. 부록에는 코드들의 github 링크와 간단한 설명 등을 첨부했다.

2. Airframe

해당 임무를 수행하기 위해 비행체는 VTOL 기능과 LiDAR, 카메라를 탑재하고 있어야한다. 이를 위해 PX4 제공 airframe인 `standard_vtol` 모델에 LiDAR와 카메라를 탑재한 모델을 만들어준다. 시뮬레이션에서 해당 모델을 활용할 수 있도록 파일 몇 개를 추가 및 수정해야 한다.



<그림 2>



<그림 3>

2-1. ~/PX4-Autopilot/Tools/simulation/gz/models/

위의 디렉토리에 Gazebo Harmonic에서 활용 가능한 모델들이 있다. 여기에 제작한 airframe의 폴더를 만들어줘야한다. 새로 만든 모델의 이름을 `standard_vtol_sensors`라고 하면, `standard_vtol_sensors/` 폴더를 추가하고 그 속에 `model.sdf`를 추가해주면 된다.

```
$ cd ~/PX4-Autopilot/Tools/simulation/gz/models/
$ mkdir -p standard_vtol_sensors/
$ cd standard_vtol_sensors
$ nano model.sdf
```

이후 <부록 1>에 있는 모델의 `.sdf` 파일을 작성하면 된다.

2-2. ~/PX4-Autopilot/ROMFS/px4fmu_common/init.d-posix/airframes

위의 디렉토리에는 PX4에서 지원하는 시뮬레이션에서 활용할 수 있는 모든 모델이 있다. 여기에 `standard_vtol_sensors`도 추가하면 된다. 해당 디렉토리에 있는 파일들은 튜닝 값을 포함하고 있다. 새로 만든 모델은 기존 `standard_vtol` 모델의 양력중심과 같은 물리적인 특성에 큰 변화가 없기 때문에 `standard_vtol`의 튜닝값을 그대로 사용해도 된다.

파일 이름은 [모델 코드]_[사용 시뮬레이션]_[모델 이름]의 규칙을 따라 짓는다. [모델 코드]는 모델 간 중복이 되면 안된다. 해당 문서에서는 혼선을 피하기 위해 Gazebo Harmonic에 부여된 코드인 `4000`번대 중 비어있는 가장 작은 숫자인 `4022`를 사용했다. [사용 시뮬레이션]은 Gazebo Harmonic의 약자인 `gz`를 작성하면 된다. 이에 따라 새로 만들 파일의 이름은 `4022_gz_standard_vtol_sensors`가 된다. 이후 짐볼을 위한 파라미터를 추가해준다. 전문은 <부록 2>에 있다.

```
$ cd ~/PX4-Autopilot/ROMFS/px4fmu_common/init.d-posix/airframes
$ cp 4004_gz_standard_vtol 4022_gz_standard_vtol_sensors
$ nano CMakeLists.txt
```

`CMakeLists.txt`에도 새로운 모델을 추가해준다. "`4021_gz_x500_flow`" 아랫줄에 방금 만든 모델 파일인 "`4022_gz_standard_vtol_sensors`"를 작성해주면 된다. 마지막으로 아래 명령을 통해 새로 만든 모델이 시뮬레이션에서 잘 생성되는지 확인해본다.

```
$ cd ~/PX4-Autopilot
$ make px4_sitl gz_standard_vtol_sensors
```

3. World

시뮬레이션과 달리 실제 환경에서는 바람과 같이 불규칙한 요인에 의해 기체가 영향 받을 수 있다. 이러한 요인에도 기체의 안정적인 제어 여부를 검증하기 위해 Gazebo에서 바람을 구현할 수 있다.

3-1. ~/PX4-Autopilot/Tools/simulation/gz/worlds/

위의 디렉토리에는 Gazebo Harmonic에서 활용할 수 있는 world 파일들이 있다. 여기에 바람이 불고 ArUco 마커가 있는 aurco_windy.sdf를 만들어준다. 아래의 명령을 통해 aruco.sdf를 복사하여 만들면 된다.

```
$ cp aruco.sdf aruco_windy.sdf
$ nano aruco_windy.sdf
```

내용 중에 <enable_wind> 객체가 있다. false를 true로 바꿔준다. 이후 <world>의 객체로 <wind>를 만들고 기본 객체인 <linear_velocity>를 활용하여 바람속도를 설정할 수 있다. 아래와 같은 모양이다. 전문은 <부록 3>에 있다.

```
<sdf>
  <world>
    …
    …
    <wind>
      <linear_velocity>5 2 0</linear_velocity>
    </wind>
  </world>
</sdf>
```

그러나 이러한 방법으로는 바람의 세기에 변화를 줄 수 없기 때문에 WindEffects 플러그인을 활용한다. 플러그인을 .sdf 코드에 포함하면 PX4와 연계해서 실행이 되지 않는다. 따라서 가제보 실행 후 플러그인을 추가한다. /usr/lib/x86_64-linux-gnu디렉토리에 아래 세 파일이 있으면 된다.

```
libgz-sim8-wind-effects-system.so
libgz-sim8-wind-effects-system.so.8
libgz-sim8-wind-effects-system.so.8.9.0
```

확인 후 가제보를 aruco_windy.sdf로 실행한다. 아래의 명령어를 활용하여 원하는 환경을 선택할 수 있다.

```
$ PX4_GZ_WORLD=aurco_windy make px4_sitl gz_<model>
```

실행하면 오른쪽에 World에 대한 설명과 아래 Entity Tree가 뜬다. 이때 World 오른쪽, Entity 1의 왼쪽에 있는 '+'를 누르면 새로운 창이 뜬다. Name에는 플러그인의 이름인 gz::sim::v8::systems::WindEffects를 작성해준다. 플러그인 이름은 아래의 명령을 통해 확인이 가능하다.

```
$ gz plugin --info --plugin libgz-sim8-wind-effects-system.so
Loading plugin library file [libgz-sim8-wind-effects-system.so]
* Found 1 plugin in library file:
  - gz::sim::v8::systems::WindEffects
* Found 3 interfaces in library file:
  - gz::sim::v8::System
  - gz::sim::v8::ISystemPreUpdate
  - gz::sim::v8::ISystemConfigure
```

Filename에서는 Wind effects를 선택한다. 리스트 가장 아래에 있다. Inner XML에서는 플러그인을 활용한 코드를 작성한다. <부록 2>의 예시와 같이 작성하면 된다.

4. 자율비행

ROS를 활용한 PX4 제어는 주로 목표 위치, 속도 등의 정보를 저장할 수 있는 trajectory_setpoint 메시지를 통해 이루어진다. 이 메시지는 CC에서 FC에게 비행체의 목표 상태에 대한 정보를 전달하며, FC는 전달받은 목표 상태를 충족하기 위해 모터 출력 등을 제어한다. 회전익 비행과 고정익 비행의 제어 방법은 다소 다르지만 VTOL은 모두 사용하므로 두 비행 방법에 대한 설명 모두 기술하였다.

trajectory_setpoint의 좌표계는 NED 기준이므로 경우에 따라서 좌표계 변환이 필요하다.

4-1. 회전익 비행

회전익 비행은 위치, 속도, 가속도와 같은 요소 중 하나의 정보만을 활용한다. 따라서 위치 기반 제어에서는 목표 위치 좌표, 속도 기반 제어에서는 목표 속도만을 제공하면 된다. 따라서 offboard_control_mode에서 제어할 요인 중 하나만 true로 설정하면 된다.

이때 위계에 따라 true로 설정된 요인 중 가장 윗 위계의 요인만 제어에 활용하며, 그 아래의 요인은 확인하지 않는다. 따라서 위치보다 아래 위계인 속도를 활용하고자 할때는 offboard_control_mode에서 position에 반드시 false를 지정해야하며, trajectory_setpoint의 position 정보에는 NaN을 부여해야한다. C++에서 NaN은 상수를 0으로 나누는 등의 방법을 활용해 만들 수 있으며, **9. 부록**의 코드에서는 quiet_NaN() 함수를 활용한다.

4-2. 고정익 비행

고정익 비행은 위치와 속도 정보를 모두 필요로 한다. 따라서 고정익 제어시 offboard_control_mode에서 position과 velocity 모두 true로 설정해야하며, trajectory_septoint에서도 position과 velocity 정보 모두 제공해야한다.

4-3. 전환(천이 및 역천이)

비행 방법을 전환하기 위해서 vehicle_command 객체 를 활용한다. 해당 메시지를 통해 기체에 특수한 명령을 전달할 수 있으며, 비행 방법 전환은 VEHICLE_CMD_DO_VTOL_TRANSITION을 활용한다. 1이 천이(회전익 → 고정익), 2가 역천이(고정익 → 회전익), 3이 회전익 비행, 4가 고정익 비행을 지칭한다. 메시지 작성은 1 또는 2를 사용할 필요 없이 vehicle_command에 VEHICLE_CMD_DO_VTOL_TRANSITION 태그와 함께 원하는 비행 방법에 해당하는 숫자(회전익: 3, 고정익: 4)를 입력하면 비행 방법의 전환이 가능하다.

천이가 계속해서 실패하는 경우 기체의 천이 시 스로틀 파라미터를 올려준다. QGC에서 Vehicle Configuration → Parameters → VTOL Attitude Control로 가서 **VT_F_TRANS_THR**를 바꾸면 된다. 그러나 QGC에서 바꿀 경우 모델 고유의 파라미터가 바뀌지 않기 때문에 시뮬레이션 실행마다 바꿔야한다. **2-2.**의 **4022_gz_standard_vtol_sensors**에서 **VT_F_TRANS_THR** 값을 바꾸면 해당 기체의 설정 값을 바꾸어 시뮬레이션 실행마다 QGC에서 바꾸는 버거움을 피할 수 있다. <부록 2>와 같이 해당 파라미터를 최댓값인 1.0으로 설정하면 천이 성공률이 높아지는 것을 확인할 수 있다.

4-4. 비행 시뮬레이션

프로그램 개발 시 하나의 기능을 구현할때마다 실행하여 검증하는 것이 좋다. <부록 2>의 비행 코드를 실행하여 VTOL 기체의 자율 비행 및 비행 방법 전환이 잘 이루어지는지 확인해보자. ROS 패키지 생성 후 실행하면 된다. 기체가 **1. 개요**의 경로를 따라 비행을 하고 시작한 위치에 착륙을 하면 작성한 프로그램이 잘 작동하는 것이다.

ROS 토픽 중 **/fmu/out/vtol_vehicle_state**를 통해 현재 비행모드를 확인할 수 있다. 해당 토픽을 확인하여 천이 및 역천이 비행이 잘 이루어지는지도 확인해보자. 비행모드는 QGC로도 확인은 가능하지만 개발하는 입장에서 ROS 토픽과 친해지면 좋다.

5. Gazebo 토픽 → ROS 토픽 변환

LiDAR 측정값을 ROS 노드에서 활용하기 위해서는 측정값이 ROS 토픽으로 발행이 되어야한다. 그러나 기체의 .sdf 파일에 추가한 센서는 Gazebo 토픽으로 발행이 된다. 따라서 Gazebo 토픽을 ROS 토픽으로 변환해야한다.

5-1. Gazebo 토픽 확인

기체 모델의 센서가 토픽을 잘 발행하는지 확인하기 위해서 아래의 명령을 수행한다.

```
$ cd PX4-Autopilot/
$ make px4_sitl gz_standard_vtol_sensors
...
Ready for takeoff!
pxh> commander takeoff
```

먼저 시뮬레이션을 실행시켜 기체를 이륙시킨다. 이후 새로운 콘솔에서 Gazebo 토픽을 확인한다.

```
$ gz topic -l
```

위의 명령은 **$ ros2 topic list**와 같이 모든 Gazebo 토픽을 나열한다. 이 중 **/world/default/model/standard_vtol_sensors_0/link/lidar_sensor_link/sensor/lidar/scan/points**라는 토픽이 있으면 해당 토픽의 내용을 확인한다.

```
$ gz topic -e -t
/world/default/model/standard_vtol_sensors_0/link/lidar_sensor_link/sensor/lidar/scan/poi
nts
```

이때 아래와 같은 구조의 내용이 뜨면 LiDAR 측정값이 잘 발행되고 있는 것이다.

```
header {
  stamp {
    ...
  }
  data {
    ...
  }
  data {
    ...
  }
}
field {
  ...
}
...
...
height: 1
width: 1
point_step: 32
row_step: 32
data: "\223B5>\000\000\000\000\000\000\000\000\000 ... \000"
is_dense: true
```

data에 해당하는 값이 변하는지 확인한다.

5-2. ROS 토픽으로 변환

아래의 명령을 실행하여 Gazebo의 LiDAR 토픽을 ROS 토픽으로 변환해주는 bridge를 구축한다.

```
$ ros2 run ros_gz_bridge parameter_bridge
/world/default/model/x500_lidar_down_0/link/lidar_sensor_link/sensor/lidar/scan/points@se
nsor_msgs/msg/PointCloud2[gz.msgs.PointCloudPacked
```

Failed to create bridge for topic ... 과 같은 메시지가 뜬 경우 ros-gz-bridge의 버전이 개발환경과 맞지 않거나 토픽 이름에 오타가 있는 등의 문제가 있는 것이니 문제를 해결하고 다시 시도해본다. 문제없이 실행이 되었으면 아래의 명령어로 ROS 토픽으로 내용 누락 없이 잘 변환되고 있는지 확인한다.

```
$ ros2 topic list
$ ros2 topic echo
/world/default/model/x500_lidar_down_0/link/lidar_sensor_link/sensor/lidar/scan/points
```

5-1. Gazebo 토픽 확인에서 확인한 것과 동일한 내용이 보이면 잘 변환되고 있는 것이다. 해당 메시지의 토픽 타입인 PointCloud2는 센서의 측정값을 data 객체에 little endian의 형식으로 발행하기 때문에 위의 토픽만으로는 데이터값이 정확한지 바로 알기 어렵다. little endian의 10진수 변환은 **6. 영상정보 처리**에서 다루는 노드인 aruco_marker에서 이루어진다.

6. 영상정보 처리

ArUco 마커 인식은 카메라의 영상정보를 처리하여 이루어진다. 이는 Python의 OpenCV 라이브러리를 활용하면 쉽게 할 수 있기 때문에 영상정보 처리를 위한 ROS 노드는 Python으로 작성했다. 시뮬레이션과 실제 기체에서 노드를 실행하게 되면 카메라 환경이 다르기 때문에 ROS 파라미터로 실행환경을 설정하게 돼있다. camera_source라는 파라미터에 0을 부여하면 물리 카메라를, 1을 부여하면 udp 포트의 카메라(시뮬레이션의 카메라)를 노드가 활용한다.

해당 노드에서는 LiDAR 데이터를 읽어 기체의 고도를 계산하는 역할도 한다. 이는 LiDAR의 측정값과 기체의 quaternion 좌표를 외적하여 계산할 수 있으며, 이를 위해서는 시뮬레이션의 LiDAR 토픽에 구독해야한다. 해당 토픽은 시뮬레이션의 환경과 기체에 따라 이름이 달라지기 때문에 각각 world와 airframe이라는 파라미터로 지정할 수 있게 되어있다. String 타입이기 때문에 파라미터 지정 시 airframe:= “standard_vtol” 과 같이 따옴표를 써줘야한다. LiDAR 토픽은 PointCloud2의 형태로 이루어져 있고, 여기서 측정값은 little endian으로 작성된다. 따라서 일련의 과정을 통해 측정 데이터를 활용이 쉬운 10진법으로 변환해준다. 이는 _lidar_cb()에 구현돼있다.

실기체의 CC는 개발환경에서 활용하는 컴퓨터보다 자원이 부족하기 때문에 프로그램 개발 시 최대한 가볍게 만들면 좋다. 이에 따라 해당 노드에서 영상은 기체의 mission_mode_가 LANDING일때만 띄우고, 이 외의 모드에서는 영상 송출을 하지 않도록 작성되어있다.

2. Airframe에서 생성한 standard_vtol_sensors 및 3. World에서 생성한 aruco_windy를 활용해 해당 노드를 실행할 경우 아래의 명령어를 사용하면 된다. 코드는 <부록 4>에서 확인 가능하다.

```
$ ros2 run imagery_processing marker_recognition --ros-args -p camera_source:=1 -p
world:= “aurco_windy”  -p airframe:= “standard_vtol_sensors”
```

실행 후 콘솔에 LiDAR 측정값을 바탕으로 계산한 고도값이 뜨는지 확인해본다. 이륙 고도와 비슷한 값이 보인면 LiDAR 측정값이 10진수로 잘 변환되어 발행되고 있는 것이다. 출력되는 내용이 없다면 ros_gz_bridge를 실행시키지 않은 등의 문제가 있는 것이니, 고도값이 제대로 출력될 때까지 문제를 찾아본다.

이후 계산된 고도가 ROS 토픽으로도 잘 발행되는지 확인해본다.

```
$ ros2 topic echo /landing/coordinates
```

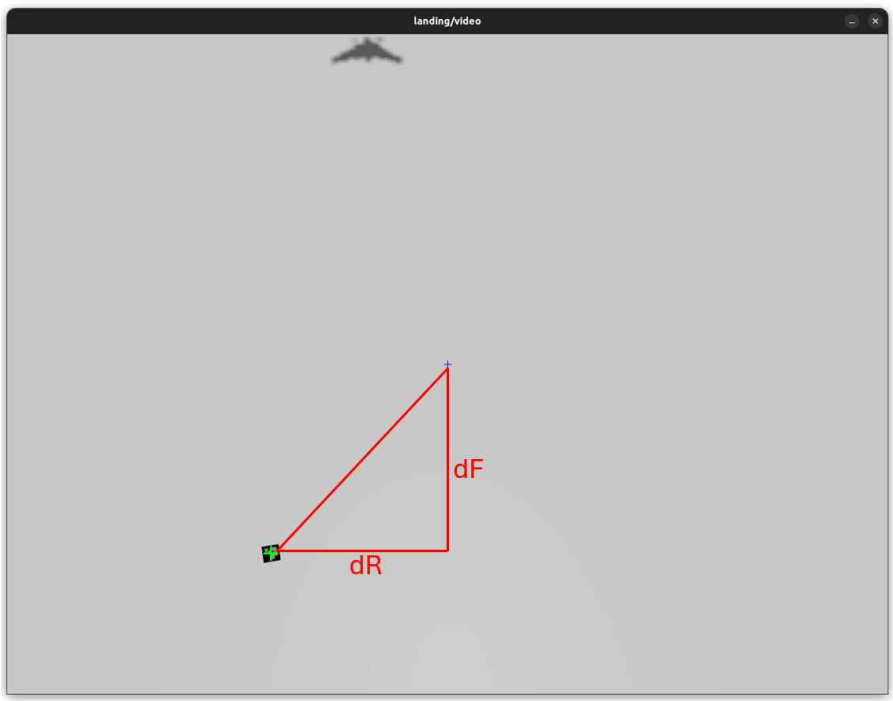
카메라 영상은 기체가 있어야 하기 때문에 정밀착륙 구현 후 확인해보거나 노드의 _mission_cb() 함수를 아래와 같이 수정한다.

```
def _mission_cb(self, msg: String) -> None:
    self._show_window = True
```

7. 정밀착륙

시뮬레이션에서는 기체가 지정한 좌표에 높은 정확도로 착륙 하지만, 실제 환경에서는 바람, 기체 진동 등에 의해 상당히 큰 오차가 발생한다. 따라서 ArUco 마커 혹은 비콘 등을 활용해 기체를 목표지점으로 유도하여 오차를 줄이고 정밀한 착륙이 가능하도록 한다. 기체의 목표 지점 유도는 크게 좌표 기준 유도와 속도 기준 유도로 나눌 수 있다.

아래는 ArUco 마커로 기체를 유도하는 화면이다. 파란색 십자가는 화면의 중앙, 초록색 십자가는 마커의 중앙으로 목표 지점이다.



<그림 4>

7-1. 좌표 기준 유도

좌표 기준 유도는 기체에게 목표 좌표를 전달해준다. 카메라에서 dF와 dR은 픽셀로 측정이 된다. `marker_recognition` 노드에서 이 값을 고도와 함께 제어 노드로 전달하면 제어 노드에서는 이를 NED 좌표로 변환 후 여기에 고도에 따라 적당한 값 k를 곱해 FC에 (목표 좌표) = (현재 좌표)+k(dN, dE)를 전달한다. 이때 k값이 클수록 과감한 제어가 이루어지지만 오차도 같이 커진다. 이러한 방법은 안정성이 높지만 하강 속력의 지정이 직관적이지 못하고 강한 바람과 같은 요인에 적극적으로 대응하지 못한다는 단점이 있다.

7-2. 속도 기준 유도

속도 기준 유도는 기체에게 목표 좌표로 갈 수 있는 속도를 전달해준다. 카메라에서 측정한 dF와 dR을 전달 받은 제어 노드는 이 값을 NED 좌표로 변환 후 단위 벡터로 변환한다. 이 좌표에 적당한 값 k를 곱해 (FC에 목표 속도) = k(dN, dE)를 전달한다. 이때 k는 기체의 목표 속력이 된다. 이러한 방법은 하강 속력의 지정이 직관적이고 갑작스러운 외부 요인에 대한 적극적인 대응에 유리하다. 그러나 좌표 기준 유도에 비해 정밀도가 떨어진다는 단점이 있다.

7-3. 착륙 시뮬레이션

<부록 5>에 코드 전문이 있다. 이를 실행하여 기체의 정밀착륙이 잘 이루어지는지 확인해본다. 사용자가 코드 수정 없이 초기 위치 및 상태를 지정할 수 있도록 파라미터를 구성했다.

`start_param`은 시작 유형을 지정한다. `0`은 사용자 수동 지정으로, 가상 환경에서는 QGC를 활용하여 특정 좌표로 기체를 이동 후 노드를 실행하면 바로 착륙을 진행한다. `1`은 사용자 지정 좌표로 자동 비행 후 착륙을 진행한다. 시작 좌표는 `start_x_param`, `start_y_param`, `start_z_param`으로 지정이 가능하며, NEU 좌표계 기준이다.

`land_param`은 착륙 시 기체 제어 방법으로 `0`은 위치 기반 제어, `1`은 속도 기반 제어이다. `descent_param`은 하강 속력으로 위치 기반 제어에서는 해당 값과 기체의 하강 속력이 일치하지 않지만, 속도 기반 제어에서는 입력 값이 하강 속력(m/s)이 된다.

아래와 같은 명령들을 통해 착륙 시뮬레이션을 실행할 수 있다. 착륙 시뮬레이션이 제대로 작동하기 위해서는 `marker_recognition` 노드와 `ros_gz_bridge` 모두 실행해야함을 잊지 말자.

```
$ ros2 run flight_control landing_test --ros-args -p start_param:=0 -p land_param:=0
-p descent_param:=1.0
```

```
$ ros2 run flight_control landing_test --ros-args -p start_param:=1 -p
start_x_param:=5 -p start_y_param:=6 -p start_z_param:=10 -p land_param:=1 -p
descent_param:=0.5
```

짐벌의 초기 위치는 앞을 바라보고 있다. QGC에서 짐볼 제어를 할 수 있다. Tilt 90°를 누르면 짐벌이 아래를 향한다.

8. 짐벌 제어

7-3. 정밀착륙을 실행하였으면 카메라가 기체의 좌표계에 고정되어 있어 기체가 움직이면 카메라도 같이 흔들려 착륙 정확도가 떨어짐을 확인할 수 있다. 따라서 기체의 자세에 따른 짐벌의 자세를 제어하여 카메라가 기체의 자세와 상관 없이 세계 좌표의 세로축에 평행하게 자세를 유지하도록 하면 착륙 정밀도를 높일 수 있을 것이다.

짐벌 통신은 Gimbal Manager라는 객체를 만들어 짐벌을 제어하는 MAVLink gimbal protocal v2를 사용한다. 과거의 프로토콜에서는 mount를, v2에서는 짐벌 매니저를 활용해 짐벌을 제어한다. 따라서 명령에 `mount` 및 `MNT`가 있으면 구형 프로토콜, `gimbal_manager` 및 `gimbalManager` 등이 있으면 v2 프로토콜을 사용함을 알 수 있다. Gimbal Manager에 한번에 하나의 MAVLink 객체에게서만 명령을 받기 때문에 기체 제어 노드에서 짐벌을 제어하기 위해서는 Gimbal Manager가 명령을 받을 객체의 `system id`와 `component id`를 지정해줘야 한다. 시뮬레이션에서는 메인 시스템과 짐벌의 시스템을 따로 나누지 않고 모두 1로 설정 돼있으며, `component id`의 경우 CC는 `1`, 짐벌은 `154`가 할당돼있다. 명령을 보내는 주체는 CC이기 때문에 `sysid/compid`는 `1/1`로 설정한다.

짐벌의 제어를 위해 Vehicle Commmand의 객체 중 아래 두 객체를 활용한다.

```
VEHICLE_CMD_DO_GIMBAL_MANAGER_CONFIGURE
VEHICLE_CMD_DO_GIMBAL_MANAGER_PITCHYAW
```

Configure 명령의 파라미터는 아래와 같다.

```
float param1; //primary control system id, in most cases 1
float param2; //primary control component id, in most cases 1
```

따라서 제어 노드에 짐벌 제어 권한 부여는 아래와 같이 할 수 있다.

```
publish_vehicle_command_(VEHICLE_CMD_DO_GIMBAL_MANAGER_CONFIGURE, 1, 1);
```

PITCHYAW의 파라미터는 아래와 같다.

```
float param1; //desired pitch [in deg]
float param2; //desired yaw [in deg]
float param3; //desired pitch rate [in deg/s]
float param4; //desried yaw rate [in deg/s]
float param5; //flags
```

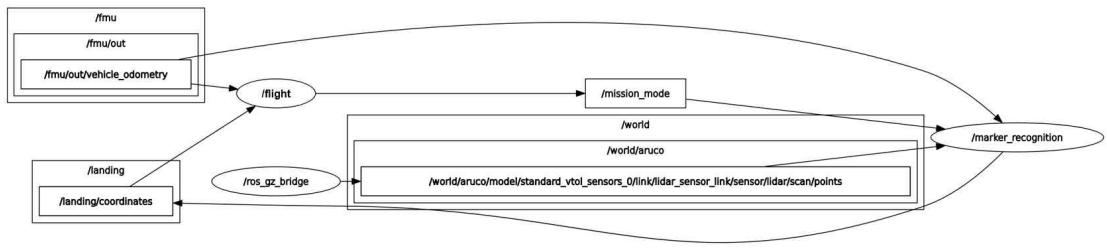
대부분의 경우 pitch와 yaw의 각도를 기준으로 짐벌을 제어하는데, 이때 **param3**과 **param4**에 아무 값도 지정하지 않으면 기본값인 **0.0**을 부여받는다. 그런데 rotaion rate이 0이면 회전을 할 수 없기에 짐벌의 자세가 바뀌지 않는다. 따라서 지정하지 않을 파라미터에 대해서는 NaN을 부여하는 것을 까먹지 말아야 원하는 대로 제어가 가능하다.

착륙을 위해 짐볼을 기본 자세인 전방(pitch = 0°, roll = 0°, yaw = 0°)에서 수직아래(pitch = −90°, roll = 0°, yaw = 0°)로 바꿔주기 위해서는 아래의 명령을 사용하면 된다.

```
publish_vehicle_command_(VEHICLE_CMD_DO_GIMBAL_MANAGER_PITCHYAW, -90, 0, nan, nan);
```

9. 통합

이제 자율 비행과 정밀 착륙을 모두 구현했으니 두 코드를 하나로 통합하여 실행해본다. <부록 6>에 통합본이 있지만 스스로 통합하여 실행해보며 디버깅하는 것을 추천한다.



<그림 5>

통합 코드를 실행하면 위와 같은 구조로 토픽과 노드가 연결된다. 이는 rqt로 확인할 수 있으며, **$ rqt**로 실행할 수 있다.

9-1. 통합 시뮬레이션 실행

시뮬레이션 실행 시 권장 순서이다. 각 단계마다 새로운 콘솔창이 필요하다.

1. uXRCEAgent 실행

```
$ MicroXRCEAgent udp4 -p 8888
```

2. QGC 실행

```
$ ./QGroundControl-x86_64.AppImage
```

3. Gazebo 실행

```
$ cd PX4-Autopilot/
$ PX4_GZ_WORLD=aruco_windy make px4_sitl gz_standard_vtol_sensors
```

4. ROS-Gazebo Bridge 구축

```
$ cd ros2_gz_ws
$ source install/setup.bash
$ ros2 run ros_gz_bridge parameter_bridge
/world/aruco_windy/model/standard_vtol_sensors_0/link/lidar_sensor_link/sensor/lidar/scan
/points@sensor_msgs/msg/PointCloud2[gz.msgs.PointCloudPacked
```

5. 영상정보 처리 노드 실행

```
$ cd ws_KRAC
$ source install/setup.bash
$ ros2 run imagery_processing marker_recognition --ros-args -p camera_source:=1 -p
world:= “aruco_windy”  -p airframe:= “standard_vtol_sensors”
```

6. 자율비행 노드 실행

```
$ cd ws_KRAC
$ source install/setup.bash
$ ros2 run flight_control landing_test --ros-args -p start_param:=2 -p land_param:=1
-p descent_param:=0.5
```

10. 부록

부록에는 해당 문서에서 활용한 코드들의 github 링크와 간단한 설명을 첨부하였다. 코드를 단순히 복사하여 활용하지 말고 이런저런 값을 변화시키며 코드를 공부하는 것을 권장한다. 또, 일부 코드는 정리가 되어있지 않은 코드이니 코드 정리와 디버깅 해보는 것 또한 추천한다.

10-1. github 링크

<부록 1> `standard_vtol_sensors/model.sdf`
https://github.com/seunguniii/space_y/blob/main/resources/standard_vtol_sensors/model.sdf
<부록 2>`4022_gz_standard_vtol_sensoros`
https://github.com/seunguniii/space_y/blob/main/resources/standard_vtol_sensors/4022_gz_standard_vtol_sensors
<부록 3> `aruco_windy.sdf`
https://github.com/seunguniii/space_y/blob/main/resources/world/aruco_windy.sdf
<부록 4> Wind Effects Plugin
https://github.com/seunguniii/space_y/blob/main/resources/world/inner.xml
<부록 5> VTOL 자율비행
https://github.com/seunguniii/space_y/blob/main/tutorial/src/flight/src/flight_test.cpp
<부록 6> 영상정보 처리 및 LiDAR 바탕 고도 계산
https://github.com/seunguniii/space_y/blob/main/tutorial/src/video/video/aruco_marker.py
<부록 7> 정밀착륙
https://github.com/seunguniii/space_y/blob/main/tutorial/src/flight/src/land_test.cpp
<부록 8> 자율 비행 및 정밀착륙
https://github.com/seunguniii/space_y/blob/main/tutorial/src/flight/src/flight.cpp

10-2. 여담

.sdf 형식은 Gazebo 모델을 위한 확장자이며, simulation designation format의 약자이다.

PX4에서 기체에 `disarm()` 명령을 전달하기 위해서는 기체의 착륙이 확인되어야 한다. 이는 `disarm()` 전에 기체에 착륙 명령을 전달하면 해결할 수 있다.

```
publish_vehicle_command(px4_msgs::msg::VehicleCommand::VEHICLE_CMD_NAV_LAND);
```

위의 명령을 빼고 바로 `disarm()` 함수를 실행하게 하면 착륙이 확인되지 않아 명령을 수행할 수 없다는 메시지를 PX4 콘솔과 QGC에서 확인할 수 있다.

아래는 <부록 6>에서 `little endian`을 10진수로 변환하는 부분을 발췌한 것이다.

```
def _lidar_cb(self, msg: PointCloud2) ->None:
        raw = bytes(msg.data)
        first_four = raw[0:4]
        self._altitude = struct.unpack('<f', first_four)[0]
        self.get_logger().info(f"calculated altitude: {self._altitude:.04f}")
```

`PointCloud2` 메시지의 `data`에서 처음 네 개의 요소를 읽은 후 `struct.unpack()` 함수를 활용해 변환한다. `<`는 기존 데이터가 `little endian`으로 이루어져있다는 뜻이고, `f`는 해당 데이터를 `float32`로 변환한다는 뜻이다.

10-3. 참고자료

PX4를 활용한 기체 제어에서 사용하는 uORB의 종류와 그에 관한 설명은 아래 링크를 참고하면 된다.

https://docs.px4.io/main/en/msg_docs/

따로 uORB 메시지로 구현돼있지 않고 vehicle_command의 객체를 활용해서 MAVLink 명령을 전달하기 위해서는 해당 객체의 파라미터를 확인해야 한다. 아래의 링크를 통해 확인이 가능하다.

https://mavlink.io/en/messages/common.html

짐벌 제어 프로토콜(Gimbal Manager)과 관련한 내용은 아래 링크를 참고하면 된다.

https://mavlink.io/en/services/gimbal_v2.html

.sdf 파일에 대한 더 자세한 내용은 아래의 링크에서 확인할 수 있다.

http://sdformat.org/

.sdf 형식에서 센서 구현에 대한 내용은 아래의 링크에서 확인할 수 있다.

https://gazebosim.org/docs/latest/sensors/

.sdf 형식에서 wind 객체에 대한 설명은 아래의 링크에서 확인할 수 있다.

http://sdformat.org/spec?ver=1.9&elem=world#world_wind

Wind Effects Plugin에 대한 설명의 아래의 링크에서 확인할 수 있다.

https://gazebosim.org/api/sim/7/classgz_1_1sim_1_1systems_1_1WindEffects.html