화재 알림 소방 드론

이선환, 이주현, 조민석, 한정훈

1. 배경

전세계적으로 화재의 피해가 급증하고 있다. 2021년 그리스, 터키, 이탈리아, 알제리 등에서는 산불이 발생하여 매우 큰 재산과 인명의 피해를 입혔고, 그리스 에비아 섬은 8월 3일날 발생한 화재로 서울시 면적의 80%에 달하는 면적이 불탔다. 터키 안탈리아주에서 시작된 불은 해당 지역에 평년보다 8배 넓은 지역을 불태우기도 했다(노예진, "'특파원보고 세계는지금' 기후변화로 인한 섭씨 50도의 폭염...최악의 산불"; "화마가 덮친 남유럽, 통제불능 기후변화"). 전문가들은 그 주요한 원인 중 하나로 기후변화를 꼽고 있다. 즉, 일시적인 것이 아니라 지속적으로 우리가 마주하고 대처해야 할 재난 중 하나가 바로 화재라는 것이다.

e-나라지표(https://www.index.go.kr)에 따르면, 2011년부터 2020년까지 우리나라는 비록 최근에 연도별 화재 발생 건수가 줄어들고 있으나, 재산 피해의 규모는 급증하고 있다.

		2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
발생건수		43,875	43,249	40,932	42,135	44,435	43,413	44,178	42,338	40,103	38,659
인명피해	소계	1,862	2,223	2,184	2,181	2,093	2,024	2,197	2,594	2,515	2,282
	사망	263	267	307	325	253	306	345	369	285	365
	부상	1,599	1,956	1,877	1,856	1840[illegible]837	1,718	1,852	2,225	2,230	1,917
재산피해		256,548	289,526	434,462	405,357	433,166	420,638	506,914	559,735	858,496	600,475

표 1 2011~2020년의 화재 현황표(단위: 건, 명, 백만원)

그림 1 화재건수 및 재산피해 현황

반면, 우리나라에서는 최근 소방공무원의 고령화가 급격히 진행되고 있다. 조성완의 연구에

따르면, 2015년 전체 소방공무원의 94,617명 중 20대와 30대가 11,971명으로 전체 4%에 불과하고 50대 이상의 소방공무원이 40,609명으로 전체 43%에 달했다. 소방공무원은 사회 재난, 자연 재난의 현장에서 국민의 생명과 재산을 보호하는 고당도의 업무를 맡고 있기에(조성완, “소방공무원의 고령화에 따른 자기효능감, 팀워크가 현장대응역량에 미치는 영향과 제도 개선방안에 대한 연구”), 멀지 않은 미래에 그 규모와 빈도가 커질 화재 재난에 효과적으로 대응할 역량이 점점 낮아지고 있다.

소방 분야에서 드론은 이러한 문제점을 해결할 하나의 주요한 대안이 될 수 있다. 화재 현장에서 현장 지휘관은 현장 대원이나 지휘 참모로부터 정보를 얻고 이로써 화재 진압을 위한 최선의 판단을 해야 한다. 그런데 현장 지휘관이 얻는 정보는 모두 음성 정보로 전달되기 때문에 지휘관 1명이 모든 정보를 통제하고 관리하기에는 어려운 한계가 있다(신열우&박진호, “경험적 접근법을 활용한 재난현장에서의 소방드론 임무수행 효율성 분석”.). 소방 드론은 화재의 범위, 연소 경로, 구대 대상자의 위치 확인 등 지상에서 놓칠 수 있는 입체적인 정보를 제공할 수 있기에(배용민, “[119기고] 소방 드론의 활용”), 드론의 투입은 화재 진압시 소방 인력을 효율적으로 활용하여 화재 진압 및 구조를 수행할 수 있다.

재난 현장에서 유용하게 사용될 수 있는데도 불구하고 현재 소방 드론의 활용 빈도는 높지 않다(배용민, “[119기고] 소방 드론의 활용”). 가장 큰 이유는 소방 드론을 운용하는 인원이 제한적이기 때문이다. 만일 사람의 전문적인 컨트롤이 필요없는 자동화 시스템을 드론이 갖춘다면, 소방대원들이 더 쉽게 드론을 화재 진압 현장에 활용할 수 있을 것이다.

전세계적으로 기후 이상 현상으로 화재 발생의 빈도 및 규모가 커지는 가운데, 소방 공무원의 노령화가 진행되는 상황 속에서 우리 프로젝트 드론팀은 소방 인력을 최소화하고 화재 진압이 효과적으로 이루어질 수 있도록 하는 화재 탐지 드론 시스템을 구축하고자 한다.

2. 개념

가) 드론: 조종사가 탑승하지 아니한 상태로 항행할 수 있는 비행체[1)]

나) 드론시스템: 드론의 비행이 유기적·체계적으로 이루어지기 위한 드론, 통신체계, 지상통제국(이·착륙장 및 조종인력을 포함한다), 항행관리 및 지원체계가 결합된 것[2)]

다) 화재: 사람의 의도에 반하거나 고의에 의해 발생하는 연소 현상으로서 소화 설비 등을 사용하여 소화할 필요가 있거나 또는 사람의 의도에 반해 발생하거나 확대된 화학적인 폭발 현상[3)]

라) 조사: 화재 원인을 규명하고 화재로 인한 피해를 산정하기 위하여 자료의 수집, 관계자 등에 대한 질문, 현장 확인, 감식, 감정 및 실험 등을 하는 일련의 활동[4)]

마) 감식: 화재 원인의 판정을 위하여 전문적인 지식, 기술 및 경험을 활용하여 주로 시각에 의한 종합적인 판단으로 구체적인 사실 관계를 명확하게 규명하는 것[5)]

바) 감정: 화재와 관련되는 물건의 형상, 구조, 재질, 성분, 성질 등 이와 관련된 모든 현상에

1) '드론 활용의 촉진 및 기반조성에 관한 법률(약칭: 드론법)', 국토교통부(첨단항공과), 법률 제16420호, 2019. 4. 30, 제정

2) '드론 활용의 촉진 및 기반조성에 관한 법률(약칭: 드론법)', 국토교통부(첨단항공과), 법률 제16420호, 2019. 4. 30, 제정

3) 소방청, 편집자. 2020년도 화재통계연감, 2021., 3쪽.

4) 소방청, 편집자. 2020년도 화재통계연감, 2021., 3쪽.

5) 소방청, 편집자. 2020년도 화재통계연감, 2021., 3쪽.

대하여 과학적 방법에 의한 필요한 실험을 행하고 그 결과를 근거로 화재 원인을 밝히는 자료를 얻는 것[6)]

사) 발화 지점: 열원과 가연물이 상호작용하여 화재가 시작된 지점[7)]

아) 화재 현장: 화재가 발생하여 소방대 및 관계자 등에 의해 소화 활동이 행하여지고 있는 장소[8)]

자) 상황실: 소방관서 또는 소방기관에서 화재·구조·구급 등 각종 소방 상황을 접수·전파 처리 등의 업무를 행하는 곳[9)]

차) 접수: 119상황실에서 화재 등의 신고를 받은 최초의 시각[10)]

카) 출동: 화재를 접수하고 119상황실로부터 출동지령을 받아 소방대가 소방서 차고에서 출발하는 것[11)]

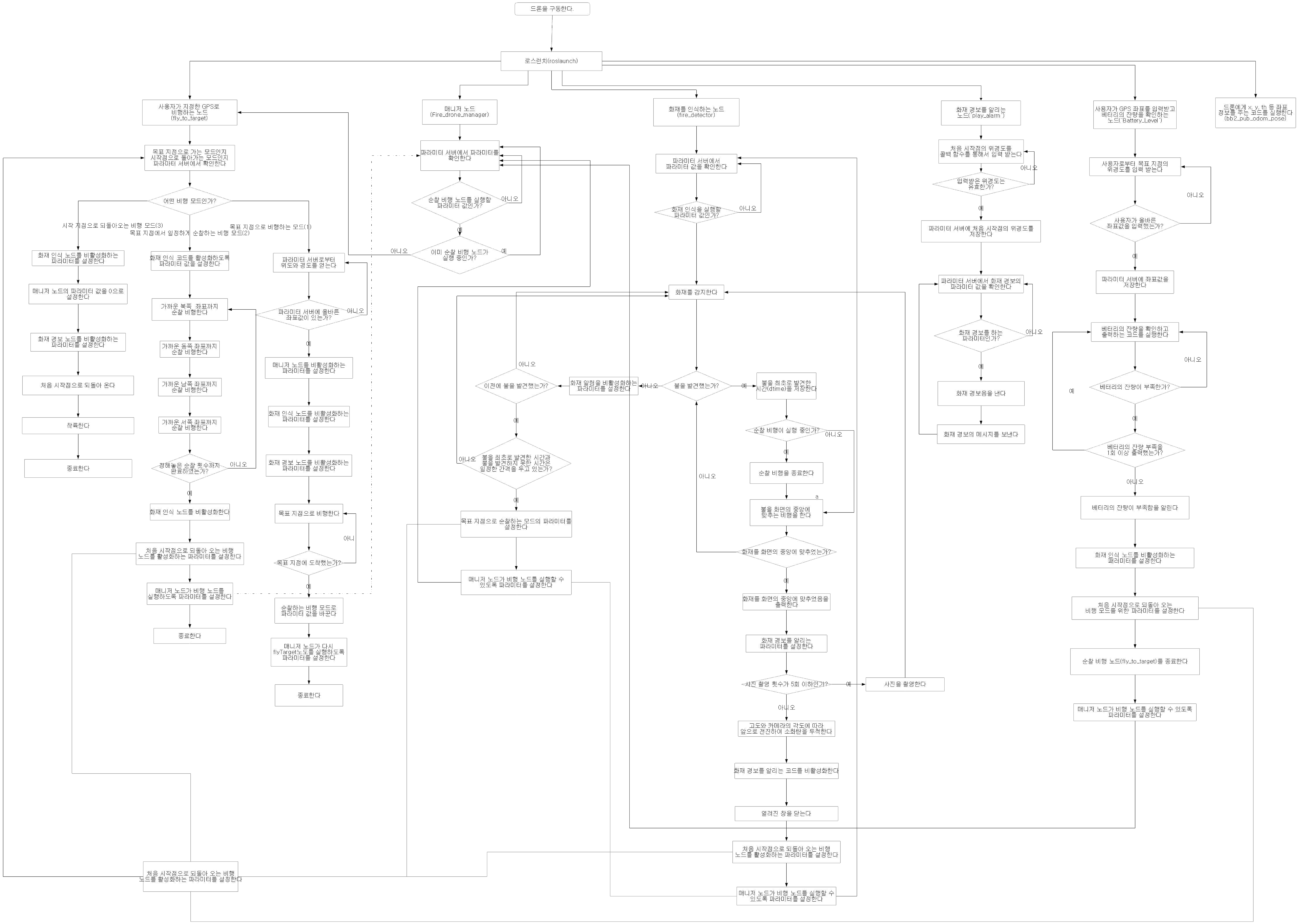
타) 도착: 출동지령을 받고 출동한 선착대가 현장에 도착하는 것[12)]

파) 잔불 감시: 화재를 진화한 화재가 재발하지 않도록 감시조를 편성하여 불씨가 완전히 소멸될 때까지 확인하는 것[13)]

---

6) 소방청, 편집자. 2020년도 화재통계연감, 2021., 3쪽.
7) 소방청, 편집자. 2020년도 화재통계연감, 2021., 3쪽.
8) 소방청, 편집자. 2020년도 화재통계연감, 2021., 4쪽.
9) 소방청, 편집자. 2020년도 화재통계연감, 2021., 4쪽.
10) 소방청, 편집자. 2020년도 화재통계연감, 2021., 4쪽.
11) 소방청, 편집자. 2020년도 화재통계연감, 2021., 4쪽.
12) 소방청, 편집자. 2020년도 화재통계연감, 2021., 4쪽.
13) 소방청, 편집자. 2020년도 화재통계연감, 2021., 4쪽.

3. 알고리즘


드론을 구동한다.
로스런치(roslaunch)
사용자가 지정한 GPS로 비행하는 노드 (fly_to_target)
목표 지점으로 가는 모드인지 시작점으로 돌아가는 모드인지 파라미터 서버에서 확인한다
어떤 비행 모드인가?
시작 지점으로 되돌아오는 비행 모드(3)
목표 지점에서 일정하게 순찰하는 비행 모드(2)
목표 지점으로 비행하는 모드(1)
화재 인식 노드를 비활성화하는 파라미터를 설정한다
매니저 노드의 파라미터 값을 0으로 설정한다
화재 경보 노드를 비활성화하는 파라미터를 설정한다
처음 시작점으로 되돌아 온다
착륙한다
종료한다
화재 인식 코드를 활성화하도록 파라미터 값을 설정한다
가까운 북쪽 좌표까지 순찰 비행한다
가까운 동쪽 좌표까지 순찰 비행한다
가까운 남쪽 좌표까지 순찰 비행한다
가까운 서쪽 좌표까지 순찰 비행한다
정해놓은 순찰 횟수까지 완료하였는가?
아니오
예
화재 인식 노드를 비활성화한다
처음 시작점으로 되돌아 오는 비행 노드를 활성화하는 파라미터를 설정한다
매니저 노드가 비행 노드를 실행하도록 파라미터를 설정한다
종료한다
파라미터 서버로부터 위도와 경도를 얻는다
파라미터 서버에 올바른 좌표값이 있는가?
아니오
예
매니저 노드를 비활성화하는 파라미터를 설정한다
화재 인식 노드를 비활성화하는 파라미터를 설정한다
화재 경보 노드를 비활성화하는 파라미터를 설정한다
목표 지점으로 비행한다
목표 지점에 도착했는가?
아니
예
순찰하는 비행 모드로 파라미터 값을 바꾼다
매니저 노드가 다시 flyTarget노드를 실행하도록 파라미터를 설정한다
종료한다
매니저 노드 (Fire_drone_manager)
파라미터 서버에서 파라미터를 확인한다
순찰 비행 노드를 실행할 파라미터 값인가?
아니오
예
이미 순찰 비행 노드가 실행 중인가?
아니오
예
이전에 불을 발견했는가?
아니오
예
불을 최초로 발견한 시간과 불을 발견하지 못한 시간은 일정한 간격을 두고 있는가?
아니오
예
목표 지점으로 순찰하는 모드의 파라미터를 설정한다
매니저 노드가 비행 노드를 실행할 수 있도록 파라미터를 설정한다
화재 알림을 비활성화하는 파라미터를 설정한다
화재를 인식하는 노드 (fire_detector)
파라미터 서버에서 파라미터 값을 확인한다
화재 인식을 실행할 파라미터 값인가?
아니오
화재를 감지한다
불을 발견했는가?
아니오
예
불을 최초로 발견한 시간(dtime)을 저장한다
순찰 비행이 실행 중인가?
아니오
예
순찰 비행을 종료한다
a
불을 화면의 중앙에 맞추는 비행을 한다
화재를 화면의 중앙에 맞추었는가?
아니오
예
화재를 화면의 중앙에 맞추었음을 출력한다
화재 경보를 알리는 파라미터를 설정한다
사진 촬영 횟수가 5회 이하인가?
예
사진을 촬영한다
아니오
고도와 카메라의 각도에 따라 앞으로 전진하여 소화탄을 투척한다
화재 경보를 알리는 코드를 비활성화한다
열려진 창을 닫는다
처음 시작점으로 되돌아 오는 비행 노드를 활성화하는 파라미터를 설정한다
매니저 노드가 비행 노드를 실행할 수 있도록 파라미터를 설정한다
화재 경보를 알리는 노드('play_alarm')
처음 시작점의 위경도를 콜백 함수를 통해서 입력 받는다
입력받은 위경도는 유효한가?
아니오
예
파라미터 서버에 처음 시작점의 위경도를 저장한다
파라미터 서버에서 화재 경보의 파라미터 값을 확인한다
화재 경보를 하는 파라미터인가?
아니오
예
화재 경보음을 낸다
화재 경보의 메시지를 보낸다
사용자가 GPS 좌표를 입력받고 베터리의 잔량을 확인하는 노드('Battery_Level')
사용자로부터 목표 지점의 위경도를 입력 받는다
사용자가 올바른 좌표값을 입력했는가?
아니오
예
파라미터 서버에 좌표값을 저장한다
베터리의 잔량을 확인하고 출력하는 코드를 실행한다
베터리의 잔량이 부족한가?
아니오
예
베터리의 잔량 부족을 1회 이상 출력했는가?
예
아니오
베터리의 잔량이 부족함을 알린다
화재 인식 노드를 비활성화하는 패러미터를 설정한다
처음 시작점으로 되돌아 오는 비행 모드를 위한 파라미터를 설정한다
순찰 비행 노드(fly_to_target)를 종료한다
매니저 노드가 비행 노드를 실행할 수 있도록 파라미터를 설정한다
드론에게 x, y, th 등 좌표 정보를 주는 코드를 실행한다 (bb2_pub_odom_pose)
처음 시작점으로 되돌아 오는 비행 노드를 활성화하는 파라미터를 설정한다

4. 세부 기능

1. 화재 감지 드론은 사용자가 정해준 일정한 영역을 체계적으로 비행한다.
2. 화재 감지 드론은 사용자가 정해준 일정한 영역에 화재가 발생했는지 여부를 판단한다.
3. 화재 감지 드론은 사용자가 정해준 일정한 영역에 발생한 화재의 위치와 규모 등을 사진 및 영상으로 촬영하여 사용자에게 즉시 알린다.

5. 코드

가) 로스 런치: 21_FireDetector.launch

```
<launch>
  <node pkg="bb2_pkg" type="21-0_Manager.py" name="Fire_drone_managerl" output="screen" >
    <param name="order" type="int" value="0" />
  </node>
  <!-- 사용자가 지정한 목표 지점으로 비행하여 순찰하는 비행 코드의 파라미터를 설정하고 실행한다. 0은 아무것도 안하는 모드, 1은 사용자가 입력한 목표 지점으로 이동하는 모드, 2는 목표지점까지 순찰하는 비행 모드, 3은 시작 지점으로 되돌아 오는 모드를 의미한다.-->
  <node pkg="bb2_pkg" type="21-1_FlyTarget.py" name="fly_to_targetl" output="screen" >
    <param name="param_of_flying" type="int" value="0" />
  </node>
  <!-- 불을 인식하는 코드의 파라미터를 설정하고 실행한다  -->
  <node pkg="bb2_pkg" type="21-2_detect_fire.py" name="fire_detectorl" output="screen" >
    <param name="param_of_detector" type="bool" value="False" />
  </node>
  <!-- 불 발생 알람을 전송하는 코드를 실행한다-->
  <node pkg="bb2_pkg" type="21-3_play_alarm.py" name="play_alarml" output="screen" >
    <param name="fire_detection_state" type="bool" value="False" />
  </node>
  <!-- 베터리의 잔량을 보여주는 노드를 실행한다 -->
  <node pkg="bb2_pkg" type="21-4_battery.py" name="Battery_Levell" output="screen" ></node>
  <!-- 드론의 xyth 등 좌표값을 발행하는 코드를 실행한다-->
  <node pkg="bb2_pkg" type="bebop_pub_odom_pose.py" name="bb2_pub_odom_posel"> </node>
  <!-- 파라미터 서버에 목적지의 위도와 경도를 각각 저장할 파라미터를 아래와 같이 설정한다-->
  <param name="tar_lati" type="double" value="0.0" />
```

```
  <param name="tar_long" type="double" value="0.0" />
  <!-- 파라미터 서버에서 처음 시작점의 위도와 경도를 저장할 파라미터를 아래와 같이 설정한다 -->
  <param name="ori_lati" type="double" value="0.0" />
  <param name="ori_long" type="double" value="0.0" />
  <!-- 순찰 비행의 횟수를 저장하는 파라미터를 아래와 같이 설정한다. -->
  <param name="patrol_num" type="int" value="3" />
</launch>
```

나) 실행 파일: 21-0_Manager.py

```
#!/usr/bin/python
#-*- coding: utf-8 -*-

#비행 및 탐지 기능을 관리하는 코드이다.

import rospy, subprocess

if __name__ == '__main__':
    rospy.init_node('Fire_drone_manager', anonymous = False)
    while not rospy.is_shutdown():
        #파라미터의 조건이 되면 코드를 실행한다.
        cod1 = rospy.get_param("/Fire_drone_managerl/order")
        if cod1 == 1:
            #순찰 비행 코드를 실행한다.
            subprocess.call(["rosrun", "bb2_pkg", "21-1_FlyTarget.py"])
            #파라미터 서버에 값을 전달하여 새로 설정되는 시간과 내부 코드의 실행 시간을 고려하여 일정한 시간을 대기하도록 한다.
            rospy.sleep(10)
        else:
            pass
```

다) 실행 파일: 21-1_FlyTarget.py

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

#FlyTarget.py는 사용자가 지정한 목적지로 가는 코드이다.

import rospy
```

```
from std_msgs.msg import Empty
from geometry_msgs.msg import Twist
from bb2_pkg.Module_Gps_Class_20 import MoveByGPS
from bb2_pkg.round_move_2 import RotateByAtti

if __name__ == '__main__':
    #노드를 초기화한다.
    rospy.init_node('fly_to_targetl', anonymous = False)
    #메시지를 초기화한다.
    tw = Twist()
    em = Empty()
    #사용자로부터 입력받은 위도, 경도를 저장할 변수를 아래와 같이 선언한다.
    target_la = 0.0
    target_lo = 0.0
    #시작점의 위도, 경도를 저장할 변수를 아래와 같이 선언한다.
    start_la = 0.0
    start_lo = 0.0
    #목적지로 비행하는 코드를 사용하기 위하여 객체를 초기화한다.
    mbg = MoveByGPS()

    #정찰 비행을 하다가 불을 인식했다가 끊긴 경우 다시 목적지로 되돌아가 순회하도록 하기 위하여 아래와 같이 반복문의 장치를 둔다.
    while not rospy.is_shutdown():
        try:
            #비행 모드의 파라미터 값을 확인한다.
            fmode = rospy.get_param("/fly_to_targetl/param_of_flying")
            #목표 지점으로 비행하는 모드라면, (즉, fmode가 1이라면)
            if fmode == 1:
                print("목표로 이동하는 모드를 시작합니다.")

                #파라미터 서버로부터 위경도의 정보를 입력받을 때까지 서버로부터 입력을 계속 입력 받는다.
                while not ( target_la >=34.0 and target_la <=38.0 and target_lo >= 126.0 and target_lo <=130.0 ):
                    if not rospy.is_shutdown():
                        target_la = rospy.get_param("/tar_lati")
                        target_lo = rospy.get_param("/tar_long")
                        print('목표 지점의 좌표를 {}, {}로 입력 받았습니다.'.format(target_la, target_lo))
                #FlyTarget이 실행 중이라면, 메니저 노드에서 다시 실행하지 않아도 되기 때문에 해당 파라미터를 0으로 초기화한다.
```

```
                rospy.get_param("/Fire_drone_managerl/order", 0)
                #목적지로 이동하기 전에 불 인식 코드를 비활성화한다
                rospy.set_param("/fire_detectorl/param_of_detector", False)
                #화재 경보를 알리는 코드를 비활성화한다.
                rospy.set_param("/play_alarml/fire_detection_state", False)
                print("설정을 완료했습니다.")
                #목적지로 이동하는 코드를 실행한다.
                mbg.fly_to_target(target_la, target_lo)
                print("도착했습니다.")
                #순찰하는 비행모드로 파라미터 값을 바꾼다.
                rospy.set_param("/fly_to_targetl/param_of_flying", 2)
                #메니저 노드에서 다시 flyTartget 노드를 실행하도록 파라미터를 설정한다.
                rospy.set_param("/Fire_drone_managerl/order", 1)
                exit()

            if fmode == 2:
                #FlyTarget이 실행 중이라면, 메니저 노드에서 다시 실행하지 않아도 되기
때문에 해당 파라미터를 0으로 초기화한다.
                rospy.get_param("/Fire_drone_managerl/order", 0)
                print("순찰 비행을 시작합니다.")
                #도착한 후에 화재를 인식하는 코드를 활성화한다.
                rospy.set_param("/fire_detectorl/param_of_detector", True)
                #목적지에서 순찰하는 코드를 실행하기 위하여 객체를 만들고 실행한다.
                print("화재 탐지를 시작합니다.")

                #순찰할 지역 지점을 얼만큼 넓게 잡을 것인가를 정하는 단위를 아래와 같
이 설정한다.
                la_unit = 0.000025
                lo_unit = 0.000050
                while not rospy.is_shutdown():
                    target_la = rospy.get_param("/tar_lati")
                    target_lo = rospy.get_param("/tar_long")
                    i = rospy.get_param("/patrol_num")
                    if i <= 0:
                        break
                    elif i%2 == 0:
                        #북쪽의 위도 좌표를 지정한다.
                        target_la = target_la + la_unit
                        #북쪽의 위도 좌표를 저장한다.
                        rospy.set_param("/tar_lati", target_la)
                        #위도의 이동 단위를 증가 시킨다
```

```
            la_unit = la_unit + la_unit
            #북쪽의 위도 좌표로 이동한다.
            mbg.fly_without_rotate(target_la, target_lo)

            #동쪽의 경도 좌표를 지정한다.
            target_lo = target_lo + lo_unit
            #동쪽의 경도 좌표를 저장한다.
            rospy.set_param("/tar_long", target_lo)
            #경도의 이동 단위를 증가시킨다.
            lo_unit = lo_unit + lo_unit
            #동쪽의 좌표로 이동한다.
            mbg.fly_without_rotate(target_la, target_lo)

        else:
            #남쪽의 위도 좌표를 지정한다.
            target_la = target_la - la_unit
            #남쪽의 위도 좌표를 저장한다.
            rospy.set_param("/tar_lati", target_la)
            #위도 단위를 증가시킨다.
            la_unit = la_unit + la_unit
            #남쪽의 좌표로 이동한다.
            mbg.fly_without_rotate(target_la, target_lo)

            #서쪽의 좌표를 지정한다.
            target_lo = target_lo - lo_unit
            #서쪽의 좌표를 저장한다.
            rospy.set_param("/tar_long", target_lo)
            #경도의 이동 단위를 증가시킨다
            lo_unit = lo_unit + lo_unit
            #서쪽의 좌표로 이동한다.
            mbg.fly_without_rotate(target_la, target_lo)

        i = i - 1
        rospy.set_param("/patrol_num", i)

    #순찰을 마친 후에 불 인식 코드를 비활성화한다.
    rospy.set_param("/fire_detectorl/param_of_detector", False)
    print("화재 탐지를 마쳤습니다.")
    #비행 모드를 되돌아오는 모드로 바꾸는 파라미터를 설정한다.
    rospy.set_param("/fly_to_targetl/param_of_flying", 3)
    #메니저 노드에서 다시 flyTartget 노드를 실행하도록 파라미터를 설정한다.
```

```
                rospy.set_param("/Fire_drone_managerl/order", 1)
                exit()

            #처음 시작점으로 되돌아오는 비행모드라면, (즉, fmode가 2라면) 아래의 코드를 실행한다.
            if fmode == 3:
                print("시작 지점으로 돌아가는 모드를 시작합니다.")
                #화재 인식 코드를 비활성화한다.
                rospy.set_param("/fire_detectorl/param_of_detector", False)
                #매니저 노드에서 다시 비행 노드를 실행하지 않도록 파라미터를 설정한다.
                rospy.set_param("/Fire_drone_managerl/order", 0)
                #화재 경보를 알리는 코드를 비활성화한다.
                rospy.set_param("/play_alarml/fire_detection_state", False)
                #파라미터 서버로부터 시작점의 위도와 경도를 가져온다.
                while not ( start_la >=34.0 and start_la <=38.0 and start_lo >= 126.0 and start_lo <=130.0 ):
                    if not rospy.is_shutdown():
                        start_la = rospy.get_param("/ori_lati")
                        start_lo = rospy.get_param("/ori_long")
                        print('시작 지점의 좌표를 {}, {}로 입력 받았습니다.'.format(start_la, start_lo))
                print("시작 지점으로 되돌아갑니다.")
                #시작점으로 되돌아 가는 비행을 한다.
                mbg.fly_to_target(start_la, start_lo)
                #착륙한다.
                mbg.mb.landing()
                #종료한다.
                exit()
        except rospy.ROSInterruptException:
            exit()
```

라) 실행 파일: 21-2_detect_fire.py

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

#detect_fire.py는 불을 발견하고 이에 따라 비행을 제어하는 코드이다.

import rospy, cv2, datetime, time, subprocess, rosnode
import serial
from std_msgs.msg import String
```

```
from bb2_pkg.MoveBB2_3 import MoveBB2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
from geometry_msgs.msg import Twist


class DetectFire:

  def __init__(self):
    #드론으로부터 이미지를 구독한다. 구독한 이미지를 콜백 함수로 보낸다.
    self.sub = rospy.Subscriber("/bebop/image_raw", Image, self.callback)
    self.bridge = CvBridge()
    self.cv_msg = cv2.imread("cimg.png")

  def callback(self,data):
    try:
      #드론으로부터 이미지 데이터를 opencv가 처리할 수 있도록 변환한다.
      self.cv_msg = self.bridge.imgmsg_to_cv2(data, "bgr8")

    except CvBridgeError as e:
      print(e)

  def save_picture(self, picture):
    try:
      img = picture
      #저장한 이미지를 시간을 접미사로 하여 명명하고 저장한다.
      now = datetime.datetime.now() #datetime.strftime(format)은 명시적인 포맷 문자열
에 의해 제어되는 날짜와 시간을 나타내는 문자열을 반환한다.
      date = now.strftime('%Y%m%d')
      hour = now.utcnow().strftime('%H%M%S%f') #화재/불의 인식은 아주 빠르게 이어
지기 때문에 초 이하 단위까지도 접사로 하여서 동일한 이름이 겹치지 않도록 한다.
      filename = '/home/iseonhwan/Documnet/ROS project/picture/fire_{}_{}.png'.fo
rmat(date, hour)
      cv2.imwrite(filename, img)
    except CvBridgeError as e:
      print(e)

if __name__ == '__main__':

    rospy.init_node('fire_detector', anonymous=False)
    fly = rospy.Publisher('/bebop/cmd_vel', Twist, queue_size = 1)
```

```
    #sp = serial.Serial('/dev/ttyUSB0', 9600)
    df = DetectFire()
    tw = Twist()
    rospy.sleep(1.0)

    mb = MoveBB2()
    #드론 비행의 스피드를 저장하는 변수
    dspeed = 0.1

    #rosnode kill 코드 블록이 조건에 따라 실행되도록 하는 인덱스 변수를 i로 설정한다. 0이면 해당 코드 블록을 실행하고, 1이면 실행하지 않는다.
    i = 0

    #이미지를 저장하는 코드의 실행 횟수를 저장하는 인덱스 변수를 j로 설정한다. 이미지를 저장하는 함수가 실행될 때마다 1씩 증가한다.
    j = 0

    #과거의 일정 시점에 불을 인지했는지 판단하는 내부 변수를 per_fire로 설정한다. 0이면, 과거의 일정 시점에 불/화재가 없었음을 의미하고 1이면, 과거의 일정 시점에 불/화재가 있었음을 의미한다.
    per_fire = 0

    #불을 인지하지 못한 시간을 저장하는 변수를 ztime으로 설정한다.
    ztime = 0.0

    #불을 인지했을 때의 시간을 저장하는 변수를 ftime으로 설정한다.
    ftime = 0.0

    try:
        #훈련 데이터로 불을 구별해낸다. CascadeClassifier()는 훈련 데이터(*.xml)로 특정 이미지의 특징을 추출해내는 함수이다. 'fire_detectionction.xml'에는 저장된 불의 특징이 저장되어 있다.
        fire_cascade = cv2.CascadeClassifier('/home/iseonhwan/Documnet/ROS project/store/fire_detection.xml')

        while not rospy.is_shutdown():
            #파라미터로부터 화재를 인식할지의 여부를 확인한다. 파라미터 '/fire_detectorl/param_of_detector'가 'True'일 경우에 조건문 아래의 화재 인식 및 조정 비행을 실행한다.
            cod1 = rospy.get_param("/fire_detectorl/param_of_detector")
            if cod1 == True:
```

```
            #드론으로부터 이미지를 불러온다.
            frame = df.cv_msg
            #비밥에서 불러온 이미지에서 불/화재 객체를 찾는다.
            fire  = fire_cascade.detectMultiScale(frame, 1.2, 5) #detectMultiScale()은
입력된 이미지에서 사이즈가 다양한 객체를 찾는 함수이다. 사각형의 리스트로서 찾은 객체가
 반환된다. 여기서 사각형의 리스트라 함은 왼쪽 꼭지점의 x좌표, y좌표, 폭(width), 넓이
(height)에 관한 값을 저장한 리스트를 말한다. 여기서는 불의 이미지를 발견하고 이를 사각
형의 리스트로서 반환된다.
             #detectMultiScale()의 매개변수는 필수적인 것 1개, 수의적인 것 5개를 취한
다. 먼저 필수적으로 객체를 찾아야 하는 공간이라 할 수 있는 이미지가 있어야 한다. 두 번
째부터는 수의적인 매개변수이다. 두 번째 매개변수는 각 이미지 범위(scale)를 얼마나 줄일
것인지 명시하는 값이고 세 번째 매개변수는 찾아낸 객체들이 얼마나 이웃하고 있어야 하는지
를 나타내는 값이며, 네 번째 매개변수는 cvHaarDetectObjects()과 같은 의미의 매개변수이
고, 다섯 번째 매개변수는 인식되는 개체가 최소 얼마나 되어야 하는지를 명시하는 값이며,
여섯 번째 매개변수는 인식되는 개체가 최대 얼마나 되어야 하는지를 명시하는 값이다.


            #fire가 0이라면, 즉, 불이 전혀 발견되지 않은 경우라면, 조건문의 코드를 실행
한다.
            if len(fire) == 0:
              #불이 발견되지 않은 경우에 "/play_alarm/fire_detection_state"를 0으로 설
정한다. /play_alarm/fire_detection_state는 화재가 발생했음을 소리나 SMS 등으로 알리는
 코드를 실행하기 위한 파라미터(parameter)이다.
                rospy.set_param("/play_alarml/fire_detection_state", False)
               #불을 잠깐 인지했으나, 화면의 중앙에 맞추는 가운데 그 객체가 사라진 경
우, 더 구체적으로 만일, 최초로 불을 발견한 시점과 불을 인지하지 못한 시점이 어느 한계를
 넘어간다면 다시 순찰 비행(Flytarget)을 실행한다.
               #그 이전에 불을 발견했으나(즉, per_fire가 1이나) 현시점에서 불이 발견되지
 않았다면 아래의 코드를 실행한다.
              if per_fire == 1:
                #불이 발견되지 않은 현재 시간을 저장한다.
                ztime = time.time()
                 #과거에 불을 발견했던 시간과 현재 불을 발견하지 못한 시간의 차이를 구
한다.
                gap_time = ztime - ftime
                 #과거의 불을 발견했던 시간과 현재 불을 발견하지 못한 시간의 차이가 10
초 이상이면(즉 그 10초 동안 불을 발견하지 못했다면) 아래의 코드를 실행한다.
                if gap_time >= 10.0:
                  print("화재로 의심되는 개체가 잠까 발견되었으나 확실하지 않습니다.")
                  #과거에 불을 감지했었음을 나타내는 인덱스 변수를 초기화한다.
                  per_fire = 0
```

```
                #이후에 다시 실행하는 detect_fire.py에서 다시 FlyTarget.py를 시작할 수 있도록 i값을 초기화한다.
                i = 0
                #사진 촬영 횟수를 초기화한다.
                j = 0
                #다시 순찰 비행을 시작하는 파마미터를 설정한다.
                rospy.set_param("/fly_to_targetl/param_of_flying", 2)
                #비행 노드를 실행하기 위하여 Manager 노드의 파라미터 값을 변경시킨다.
                rospy.set_param("/Fire_drone_managerl/order", 1)
            else:
              #불 또는 화재가 발견된 경우라면 아래의 코드를 실행한다.
              for (x,y,w,h) in fire:
                #리스트 fire의 값만큼 아래의 코드를 실행한다.
                #리스트 fire의 값을 활용하여 이미지에 사각형을 그린다.
                cv2.rectangle(frame,(x-20,y-20),(x+w+20,y+h+20),(255,0,0),2)

                print("화재가 의심되는 장면을 포착하였습니다.")
                #일단 불을 발견했다면, 불을 인지했는지 판단하는 내부 변수(per_fire)를 변화시킨다.
                per_fire = 1

                #불을 발견했다면, 불을 인지한 시간을 저장한다.
                ftime = time.time()

                #불을 발견했다면 패트롤 모드(patrol mode)를 종료하고 adjust mode를 실행한다. 단, 불을 발견한 반복문 아래에서 한 번만 실행한다. 이를 위하여 인덱스 변수 i를 활용한다.
                if i == 0:
                  rosnode.kill_nodes(['/fly_to_targetl'])
                  i = 1

                #사각형의 가운데 좌표를 구한다.
                fx = x + (w * 0.5)
                fy = y + (h * 0.5)

                #인식된 불이 화면의 중앙에 있을 경우에는 아래의 코드를 실행한다.
                if ((fx >= 190)and(fx <= 450)) and ((fy >= 150)and(fy <= 330)):
                  #드론의 비행을 멈춘다.
                  tw.linear.x = 0
                  tw.linear.y = 0
```

```
                fly.publish(tw)
                print("화재 장면을 화면의 중앙에 맞췄습니다({}, {})".format(fx, fy))
                #화재 경보를 알리는 코드를 활성화한다.
                rospy.set_param("/play_alarml/fire_detection_state", True)
                #동일한 화재에 대한 사진 촬영의 횟수가 5미만이면 사진을 찍는다.
                if j < 5:
                    #인식한 화재/불의 이미지를 저장한다.
                    df.save_picture(frame)
                    print("화재 장면을 {}회 찍었습니다.".format(j+1))
                    #사진 촬영의 횟수를 j에 반영한다.
                    j = j + 1
                else:
                    #사진을 일정 횟수까지 저장했다면, 드론이 일정 거리 앞으로 이동하여 소화탄을 투척한다. 여기서 일정 거리란, 드론이 화재를 인식한 각도, 그리고 드론의 높이를 고려한 거리이다. 드론이 3m 높이에서, 높이를 기준으로 45도 꺾어서 화재를 탐지하는 것을 전제로 함으로 앞으로 3m 이동하여 소화탄을 투척한다.
                    mb.move_x(3, 0.01)
                    sp.write('1')
                    print("소화탄을 투척합니다.")
                    #소화탄을 투척했다면, 화재 경보를 알리는 코드를 비활성화한다.
                    rospy.set_param("/play_alarml/fire_detection_state", False)
                    #처음 지점으로 되돌아 오는 비행 모드의 파라미터를 설정한다.
                    rospy.set_param("/fly_to_targetl/param_of_flying", 3)
                    #매니저 노드가 비행 코드를 실행하도록 파라미터를 설정한다
                    rospy.set_param("/Fire_drone_managerl/order", 1)
                    rospy.sleep(2)
                    #열려진 창을 닫는다.
                    cv2.destroyWindow('frame')
                    exit()

            #화재가 화면의 아래에서 발견된 경우 뒤로 이동시킨다.
            elif ((fx >= 190) and (fx <= 450)) and ((fy >= 331) and(fy <= 480)):
                tw.linear.x = -dspeed
                tw.linear.y = 0
                fly.publish(tw)

            #화재가 화면의 위에서 발견된 경우 드론을 앞으로 이동시킨다.
            elif ((fx >= 190) and (fx <= 450)) and ((fy >= 0 ) and(fy <= 149)):
                tw.linear.x = dspeed
                tw.linear.y = 0
                fly.publish(tw)
```

```
            #화재가 화면의 왼쪽에서 발견된 경우 드론을 왼쪽으로 이동시킨다.
            elif ((fx >= 0) and (fx <= 189)) and ((fy >= 151 ) and(fy <= 329)):
                tw.linear.x = 0
                tw.linear.y = dspeed
                fly.publish(tw)

            #화재가 화면의 왼쪽 아래에서 발견된 경우 드론을 왼쪽 아래로 이동시킨
다.
            elif ((fx >= 0) and (fx <=189)) and ((fy >= 331) and (fy <= 480)):
                tw.linear.x = -dspeed
                tw.linear.y = dspeed
                fly.publish(tw)

            #화재가 화면의 왼쪽 위에서 발견된 경우 드론을 왼쪽 위로 이동시킨다.
            elif ((fx >= 0) and (fx <=189)) and ((fy >= 0) and (fy <= 149)):
                tw.linear.x = dspeed
                tw.linear.y = dspeed
                fly.publish(tw)

            #화재가 오른쪽 아래에서 발견된 경우 드론을 오른쪽 아래로 이동시킨다.
            elif ((fx >= 450) and (fx <=640)) and ((fy >= 331) and (fy <= 640)):
                tw.linear.x = -dspeed
                tw.linear.y = -dspeed
                fly.publish(tw)

            #화재가 오른쪽 위에서 발견된 경우, 드론을 오른쪽 위로 이동시킨다.
            elif ((fx >= 450) and (fx <=640)) and ((fy >= 0) and (fy <= 149)):
                tw.linear.x = dspeed
                tw.linear.y = -dspeed
                fly.publish(tw)

            #화재가 오른쪽에 있을 경우, 드론을 오른쪽으로 이동시킨다.
            elif ((fx >= 451) and (fx <= 640)) and ((fy >= 150 ) and(fy <= 330)):
                tw.linear.x = 0
                tw.linear.y = -dspeed
                fly.publish(tw)
            else:
              pass

        cv2.imshow('frame', frame)
```

```
            #waitKey()는 키 입력을 기다리는 대기 함수이다. 매개변수는 대기 시간인데, 단위는 ms(millisecond, 1ms = 0.001)이다. 0xFF는 255를 의미한다. cv2.waitKey(1)의 반환값은 -1이고, -1과 255를 &연산하면 255가 된다. 그리고 ord()는 문자의 유니코드 값을 돌려주는 함수이다. ord('q')의 반환값은 113이다. 즉, 이 조건문은 결코 참이 될 수 없으므로 어떤 키 입력을 받더라도 위의 조건문은 멈추지 않는다.
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break

        rospy.spin()

    except KeyboardInterrupt:
        print("Shutting down")
```

마) 실행 파일: 21-3_play_alarm.py

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

#play_alarm.py는 시작 지점의 위도와 경도를 파라미터에 저장하고, 화재가 발생했을 경우 특정 사용자(들)에게 화재 발생 여부 및 그 지역을 알리는 프로그램이다.

from twilio.rest import Client
import rospy
from playsound import playsound
from bebop_msgs.msg import Ardrone3PilotingStatePositionChanged

#cb_get_gps()는 드론의 토픽으로부터 위도와 경도의 정보을 구독할 때, 실행할 콜백 함수이다.
def cb_get_gps(msg):
    global lati_now
    global long_now
    lati_now = msg.latitude
    long_now = msg.longitude

if __name__ == '__main__':
    rospy.init_node('play_alarm')
    #드론으로부터 드론의 gps값을 구독하고 그 값을 콜백 함수로 처리한다.
    rospy.Subscriber('/bebop/states/ardrone3/PilotingState/PositionChanged', Ardrone3PilotingStatePositionChanged, cb_get_gps)
    #화재 발생 여부 및 그 지역 정보를 문자로 받기 위해 필요한 정보를 저장하고 처리한다.
    account_sid = 'AC3a674bf50d4d0511d8600550e6e50739'
```

```
    auth_token = '715e9cbd3d7bbd699606e9915362381c'
    client = Client(account_sid, auth_token)

    #일단, 구동하자마자 시작점의 위도, 경도를 파라미터에 저장한다.
    while not rospy.is_shutdown():
        try:
            #콜백 함수로부터 받은 위도와 경도의 정보가 유효하다면 이를 파라미터에 저장한다. 유효하다는 의미는 위도와 경도의 범위가 대한민국의 지역 안에 해당될 수 있음을 나타낸다.
            if ( lati_now >=34.0 and lati_now <=38.0 and long_now >= 126.0 and long_now <=130.0 ):
                start_lati = lati_now
                start_long = long_now
                rospy.set_param("/ori_lati", start_lati)
                rospy.set_param("/ori_long", start_long)
                break
        except:
            continue

    try:
        while not rospy.is_shutdown():
            #파라미터로부터 화재 경보를 울려야 하는지 확인하고 만일, 화재를 경보를 울려야 한다면 조건문 아래의 코드를 실행한다.
            if rospy.get_param("/play_alarml/fire_detection_state") is True:
                #화재 경보음을 울린다.
                playsound('/home/iseonhwan/Documnet/ROS project/store/alarm.mp3')
                #화재 경보 메시지를 보낸다.
                alarm_s1 = str(lati_now) + ', '
                alarm_s2 = str(long_now)
                alarm_s3 = "에서 화재가 발생하였습니다"
                alarm_message = alarm_s1+alarm_s2+alarm_s1+alarm_s3
                print(alarm_message)
                #사용자의 번호로 화재 정보의 메시지를 보낸다.
                message = client.api.account.messages.create(to="+821077572419", from_="+17378885431", body= alarm_message )
                rospy.sleep(3)
    except rospy.ROSInterruptException:
        exit()
```

바) 실행 파일: 21-4_battery.py

```
#!/usr/bin/python
#-*- coding: utf-8 -*-

# battery.py는 사용자로부터 목적지의 위도와 경도 정보를 입력받고, 드론의 배터리 정보를 확인하는 프로그램이다.

import rospy, subprocess, rosnode
from bebop_msgs.msg import CommonCommonStateBatteryStateChanged as Battery
# 비밥 드론의 메세지 토픽을 받아옴

#callback()은 드론으로부터 구독하는 배터리 정보를 처리하는 콜백 함수이다.
def callback(data):
    global battery_percent
    battery_percent = int(data.percent)

if __name__ == '__main__':

    rospy.init_node('Battery_Level', anonymous = False)
    #드론으로부터 배터리 정보를 구독하고 그 정보를 콜백 함수로 넘긴다.
    rospy.Subscriber("bebop/states/common/CommonState/BatteryStateChanged", Battery, callback)

    while not rospy.is_shutdown():
        try:
            #사용자로부터 목적지의 위도와 경도를 입력 받는다. 입력의 편의를 위하여 한 줄로 입력받고, 그 메시지의 처리는 아래와 같이 자동으로 처리한다.
            inmessage = str(input("위도와 경도를 쉼표로 구분하여 입력하세요:"))
            inmessage = inmessage.replace(" ", "")
            inmessage = inmessage.replace("(", "")
            inmessage = inmessage.replace(")", "")
            slati = inmessage.split(",")[0]
            slong = inmessage.split(",")[1]
            target_la = float(slati)
            target_lo = float(slong)
            #위도와 경도가 유효한 숫자일 경우에 입력받은 위도, 경도를 파라미터 서버에 저장하고 다음 단계로 넘어간다.
            if ( target_la >=34.0 and target_la <=38.0 )and( target_lo >= 126.0 and target_lo <=130.0 ):
                #입력받은 위경도를 파라미터 서버에 저장한다.
                rospy.set_param("/tar_lati", target_la)
```

```
                rospy.set_param("/tar_long", target_lo)
                #목표 지점으로 이동하는 비행 모드의 파라미터를 설정한다.
                rospy.set_param("/fly_to_targetl/param_of_flying", 1)
                #반복문을 빠져 나간다.
                break
            else:
                print("다시 입력해주세요.")
        except:
            print("오류입니다. 다시 입력하세요.")
            continue

    #반복문 안에서 특정 조건문의 코드 블록을 한 번만 실행하고 다시 실행하지 않도록 하기 위하여 인덱스 변수 i를 설정한다. 여기서 특정 조건문의 코드 블록이란,
    i = 0
    while not rospy.is_shutdown():
        try:
            rospy.loginfo("배터리 잔량: %d%% ", battery_percent)
            rospy.sleep(20)
            if (i == 0) and (battery_percent < 15) and (rospy.get_param("/fire_detectorl/param_of_detector") != 3):
                #베터리의 잔량을 터미널에서 출력한다.
                print("베터리 잔량이 15% 이하 입니다. 되돌아 갑니다.")
                #불 인식 코드를 비활성화한다.
                rospy.set_param("/fire_detectorl/param_of_detector", False)
                #처음 시작점으로 되돌아 오는 비행 코드를 활성화한다.
                rospy.set_param("/fly_to_targetl/param_of_flying", 3)
                #현재 실행 중인 비행 코드를 종료한다. rosnode.kill_nodes()는 특정 노드를 죽이는 함수이다. 매개변수로 종료시킬 노드의 이름을 취한다.
                rosnode.kill_nodes(['/fly_to_targetl'])
                rospy.sleep(3)
                #다시 비행 코드를 실행하도록 메니저의 파라미터를 설정한다.
                rospy.set_param("/Fire_drone_managerl/order", 1)
                rospy.sleep(3)
                i = 1
        except:
            continue
```

사) 모듈 파일: Module_Gps_Class_20.py

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
```

```
#표준 라이브러리를 임포트한다.
import rospy, sys

from scipy import sqrt, cos, sin, arctan2, pi
from math import degrees, radians, sqrt
from haversine import haversine

#필요한 메시지를 임포트한다.
from std_msgs.msg import Empty
from geometry_msgs.msg import Twist
from bebop_msgs.msg import Ardrone3PilotingStateAttitudeChanged
from bebop_msgs.msg import Ardrone3PilotingStatePositionChanged
from bebop_msgs.msg import Ardrone3PilotingStateAltitudeChanged
from bebop_msgs.msg import Ardrone3GPSStateNumberOfSatelliteChanged
from bebop_msgs.msg import CommonCommonStateBatteryStateChanged as Battery

#사용자가 만든 클래스를 임포트한다.
from bb2_pkg.MoveBB2_3 import MoveBB2

'''
    GPS for center of map   ( 36.51994848698016, 127.17306581466163)
    Parot-Sphinx start GPS ( 48.878900,              2.367780          )
    diffrence                  (-12.358951513,      +124.805285815      )
'''

#직선으로 움직이는 속도값을 일정하게 적용하기 위하여 전역 변수, LIN_SPD를 만들고, 0.30으로 정해준다.
LIN_SPD     = 0.30
#회전하는 속도값을 일정하게 적용하기 위하여 전역 변수, ANG_SPD를 만들고 그 값을 0.30으로 정해준다.
ANG_SPD     = 0.30 * pi
#비행 고도을 일정하게 적용하깅 위하여 전역 변수, FLIFHT_ALT를 만들고, 그 값을 4(m)로 정해준다.
FLIGHT_ALT = 4
#각도의 회전 1도의 거리가 얼만큼인지를 아래의 값으로 정한다.
DEG_PER_M   = 0.00000899320363721

'''
                                p2 (lat2,lon2)
                        | | |   /
```

```
                    | | |  /
                    | | |0/
                    | | |/
                    | |0/
                    | |/
                    |0/<--- bearing
                    |/--------
                    p1 (lat1,lon1)
```

when center is (a,b), equation of circle : pow((x-a),2) + pow((y-b),2) = pow(r,2)
중심이 (a, b)에 있을 때, 원의 방정식: pow((x-a), 2) + pow((y-b), 2) = pow(r, w)
pow()는 파이썬에서 math의 매서드 함수, 파이썬 내장 함수 두 가지가 있다. 파이썬 내장 함수로서의 pow()는 제곱을 구하는 함수이다. 즉, pow(x, y)는 x(밑, base)의 y제곱(지수, exp)을 계산해서 반환한다. 내장 함수 pow()는 인자들의 자료형에 따라 Integer, Float, Complex nuber 등으로 결정된다. 반면, math.pow()는 두 인자를 float형으로 바꾼다. (출처: AON. "(PYTHON) pow() vs math.pow()". AttackOnNunu, 2020년 3월 2일
. https://taejin0527.github.io/[object Object]/Language-Python-20200302-PY-pow-vs-mathpow/index.html.)
'''

```
class MoveByGPS:

    def __init__(self):
        #드론의 비행 자세 변화(Drones attitude changed)에 대한 토픽을 구독한다. 메시지에는 롤(roll), 피치(pitch), 야우(Yaw) 값이 척도는 라디안(radian) 변수형은 float32로 담겨 있다.
        rospy.Subscriber('/bebop/states/ardrone3/PilotingState/AttitudeChanged',
                         Ardrone3PilotingStateAttitudeChanged,
                         self.cb_get_atti)
        #드론의 위치 변화(Drones position changed)에 대한 토픽을 구독한다. 메시지에는 위도값(Latitude position), 경도값(Longitude position), 고도값(Altitude)이 담겨 있다. 위도와 경도의 척도는 십진법 각도이고 사용할 수 없는 경우 500.0으로 되돌려준다. 고도값의 척도는 미터(meter)이다.
        rospy.Subscriber('/bebop/states/ardrone3/PilotingState/PositionChanged',
                         Ardrone3PilotingStatePositionChanged,
                         self.cb_get_gps)
        #드론의 고도 값(Drones altitude changed.n)에 대한 토픽을 구독한다. 고도값은 이륙한 지점부터 보고된다. 메시지는 float64의 고도값을 담고 있다.
        rospy.Subscriber('/bebop/states/ardrone3/PilotingState/AltitudeChanged',
```

```
                        Ardrone3PilotingStateAltitudeChanged,
                        self.cb_get_alti)
        #GPS 위성의 수에 대한 토픽을 구독한다. 자료형은 uint8이다.
        rospy.Subscriber('/bebop/states/ardrone3/GPSState/NumberOfSatelliteChanged',
                        Ardrone3GPSStateNumberOfSatelliteChanged,
                        self.cb_get_num_sat)
        #드론의 배터리 상태(Battery state)에 대한 토픽을 구독한다. 배터리가 얼마나 충전되어 있는지가 백분율(percentage)로 저장되어 있다. 자료형은 uint8이다.
        rospy.Subscriber("bebop/states/common/CommonState/BatteryStateChanged", Battery, self.cb_bt_check)

        #클래스 내의 스태택 변수를 아래와 같이 정의한다.
        self.atti_now = 0.0
        self.atti_tmp = 0.0
        #use_tmp는 get_atti() 및 rotate() 등에서 특정한 조건문이 실행될 수 있도록 하는 기준값을 저장한 변수이다.
        self.use_tmp = False
        #위도값
        self.lati_now = 1000.0
        #경도값
        self.long_now = 1000.0
        #절대적인 고도값
        self.alti_gps = 0.0
        #(이륙 지점을 0으로 하는) 상대적인 고도값
        self.alti_bar = 0.0

        self.bearing_now = 0.0

        self.bearing_ref = 0.0
        #배터리를 상태의 값을 담는 스태틱 변수
        self.battery_percent = 100.0

        self.margin_angle  = radians(5.0)
        #반지름의 오차 범위를 DEG_PER_M * 1.5로 정의한다.
        self.margin_radius = DEG_PER_M * 1.5
        #고도값의 오차 범위를 변수명 margin_alt, 값은 0.25로 정의한다.
        self.margin_alt    = 0.25

        self.tw = Twist()
        #비밥을 움직이는 코드의 객체를 만든다.
```

```
        self.mb = MoveBB2()

        rospy.sleep(3.0)

    #cb_bt_check()는 배터리 토픽을 구독할 때 실행되는 콜백 함수이다.
    def cb_bt_check(self, data):
        #토픽으로부터 받은 data를 int형으로 바꾸어 battery_percent에 저장한다.
        self.battery_percent = int(data.percent)

    #cb_get_num_sat()는 위성의 수를 구독할 때 실행되는 콜백 함수이다.
    def cb_get_num_sat(self, msg):
        pass

    #cb_get_alti()는 드론의 고도값을 구독할 때 발생하는 콜백 함수이다.
    def cb_get_alti(self, msg):
        self.alti_bar = msg.altitude

    #cb_get_gps()는 드론의 GPS 위치값에 대한 토픽을 구독할 때 발생하는 콜백 함수이
다.
    def cb_get_gps(self, msg):
        self.lati_now = msg.latitude  + OFFSET_LAT
        self.long_now = msg.longitude + OFFSET_LON
        self.alti_gps = msg.altitude

    #cv_get_atti()는 드론의 자세 변화에 대한 토픽을 구독할 때 작동하는 콜백 함수이다.
    def cb_get_atti(self, msg):
        #드론의 야우(yaw)값을 atti_now에 일단 저장한다.
        self.atti_now = msg.yaw
        #야우값이 0보다 작을 경우,
        if    msg.yaw < 0:
            #msg.yaw에 파이(pi, 원주율, 3.141592653589793)를 더하여 atti_tmp에 저장
한다. 라디안 척도에서 파이 라디안은 180도를 의미한다.
            self.atti_tmp = msg.yaw + pi
        #야우값이 0보다 클 경우,
        elif msg.yaw > 0:
            #msg.yaw에 파이를 빼서 atti_tmp에 저장한다.
            self.atti_tmp = msg.yaw - pi
        #그 외의 경우(즉, yaw가 0일 경우)
        else:
            self.atti_tmp = 0.0
```

```
    #get_atti()는 특정한 조건에서 atti_tmp 혹은 atti_now를 반환하는 함수이다.
    def get_atti(self):
        #use_tmp의 값이 참일 경우
        if self.use_tmp == True:
            #atti_tmp를 반환한다. atti_tmp는 파이 라디안이 더해지거나 빼진 야우값이
다.
            return self.atti_tmp
        #use_tmp의 값이 참이 아닐 경우(즉, False인 경우)
        else:
            #atti_now를 반환한다.
            return self.atti_now

    #get_bearing()은 시작점(lat1, lon1)과 목표점(lat2, long2)의 gps의 두 지점의 각도를
구하는 함수이다.
    def get_bearing(self, lat1, lon1, lat2, lon2):
        #위도와 경도의 값을 라디안으로 변환하여 저장한다.
        Lat1,  Lon1 = radians(lat1), radians(lon1)
        Lat2,  Lon2 = radians(lat2), radians(lon2)

        #경도간의 차이(즉, 거리)의 사인(sin) 값을 구한고, 또 목표 지점의 위도값의 코사인
 값을 구한다.
        #참고할 자료: https://tttsss77.tistory.com/177
        y = sin(Lon2-Lon1) * cos(Lat2)
        x = cos(Lat1) * sin(Lat2) - sin(Lat1) * cos(Lat2) * cos(Lon2-Lon1)

        #x, y에 대한 아크탄젠트를 구하고 이를 반환한다.
        return arctan2(y, x)


    #get_gps_now()는 GPS로부터 얻은 현재 위도와 경도의 값을 반환하는 함수이다.
    def get_gps_now(self):
        return self.lati_now, self.long_now

    #rotate()는 목표 지점을 정면으로 향하도록 하는 함수이다.
    def rotate(self, lat2, lon2, speed):

        pub = rospy.Publisher('/bebop/cmd_vel', Twist, queue_size = 1)
        tw  = Twist()
        #get_gps_now()로부터 현재의 위도와 경도 값을 얻는다.
        lat1, lon1 = self.get_gps_now()
        #현재의 위도와 경도 값과 목표 지점의 위도와 경도 값을 인자를 get_bearing()에
```

```
전달하고 그 반환값을 target에 저장한다.
        target  = self.get_bearing(lat1, lon1, lat2, lon2)
        #atti_now값을 current에 저장한다. atti_now는 드론의 야우(yaw)값을 저장하고 있
다.
        current = self.atti_now
        #현재의 각도와 목표 지점과의 상대적 각도의 차이를 변수 angle에 저장한다.
        angle = abs(target-current)

        #angle이 파이보다 클 경우, 즉 180도보다 클 경우
        if angle > pi:
            #use_tmp에 True를 저장한다.
            self.use_tmp = True
            #target이 0.0보다 클 경우
            if   target > 0.0:
                #target의 값에 파이를 뺀다.
                target = target - pi
            #target이 0.0보다 작을 경우
            elif target < 0.0:
                #target의 값에 파이를 더한다.
                target = target + pi
            else:   pass
             #get_atti()의 반환값을 current에 저장한다. get_atti()는 use_tmp가 True이
냐, False이냐에 따라 attim_tmp, atti_now를 반환한다.
            current = self.get_atti()
            angle = abs(target - current)
        #angle이 파이보다 작을 경우에, 즉 180도보다 작을 경우
        else:
            #use_tmp에 False를 저장한다.
            self.use_tmp = False

        #target이 current보다 클 경우,
        if   target > current:
            #드론이 시계 방향으로 돌도록 음수의 값을 준다.
            tw.angular.z = -speed

            if   angle > radians(50):
                target = target - radians(5)
            elif angle > radians(20):
                target = target - radians(10)
            else:
                tw.angular.z = -0.1125
```

```
            #target이 current보다 큰 동안에
            while target > current:
                #만일 드론의 회전값이 0.125보다 클 경우에
                if abs(tw.angular.z) > 0.125:
                    #target과 current의 차이가 어느 정도 큰가에 따라 비례적으로 회전
값을 준다.
                    tw.angular.z = -speed * abs(target - current) / angle
                else:
                    tw.angular.z = -0.125
                #get_atti()의 변환값을 다시 current에 저장한다. get_atti는 use_tmp의
값에 따라서 (yaw값과 같은) atti_now 또는 (yaw값에 pi를 더하거나 뺀) atti_tmp를 반환한
다.
                current = self.get_atti()
                #tw를 발행하여 드론을 회전시킨다.
                pub.publish(tw)
        #target이 current보다 작을 경우 , 위의 과정을 동일하게 하되 단 드론의 회전값을
 양수로 한다.
        elif target < current:
            #시계 반대 방향으로 돌도록 양수의 값을 준다.
            tw.angular.z =  speed

            if   angle > radians(50):
                target = target + radians(5)
            elif angle > radians(20):
                target = target + radians(10)
            else:
                tw.angular.z =  0.1125

            while target < current:
                if abs(tw.angular.z) > 0.125:
                    tw.angular.z =  speed * abs(target - current) / angle
                else:
                    tw.angular.z =  0.125
                current = self.get_atti();  pub.publish(tw)

        else:   pass

    #check_route()는 현재 지점과 목표 지점의 각도가 일정 범위 안에 있는지의 여부를 알
리는 함수이다.
    def check_route(self, lat2, lon2):
```

```
        #get_gps_now()의 반환값을 lat_now, lon_now에 저장한다. get_gps_now()는 현재
의 위도, 경도 값을 반환한다.
        lat_now, lon_now = self.get_gps_now()
        #get_bearing에 현재의 위도, 경도, 그리고 매개변수의 위도, 경도를 인자로 주고 그
 반환값을 bearing에 저장한다. 즉, gps 상 두 지점의 각도를 brearing에 저장한다.
        bearing = self.get_bearing(lat_now, lon_now, lat2, lon2)
        #bearing의 값이 self.bearing_ref를 기준으로 self.margin_angle의 범위 밖이라면
        if bearing > self.bearing_ref - self.margin_angle and \
           bearing < self.bearing_ref + self.margin_angle:
            #True(참)를 반환한다.
            return True
        else:
            #False(거짓)를 반환한다.
            return False

    #check_alt()는 현재의 (상대적) 고도가 사용자가 지정한 일정한 범위에 있는지를 알리는
 함수이다.
    def check_alt(self):
        #만일, self.alti_bar가 목표의 고도를 기준으로 self.margin_alt의 범위 안에 있다
면,
        if self.alti_bar > FLIGHT_ALT - self.margin_alt and \
           self.alti_bar < FLIGHT_ALT + self.margin_alt:
            #True(참)를 반환한다.
            return True
        #self.alti_bar가 목표의 고도를 기준으로 self.margin_alt의 범위 밖이라면,
        else:
            #False(거짓)을 반환한다.
            return False

    #check_arrived()는 목표의 좌표에 도착했는지의 여부를 판별하는 함수이다.
    def check_arrived(self, lat2, lon2):
        #get_gps_now()를 실행하여 그 반환값을 lat_now, lon_now에 저장한다. 즉, 현재
위도와 경도의 값을 저장한다.
        lat_now, lon_now = self.get_gps_now()
        #현재의 위도와 목표의 위도 차이, 현재의 경도와 목표의 경도 차이를 이용하여 삼각
함수로써 거리를 구한다.
        radius = sqrt(pow((lat_now-lat2), 2) + pow((lon_now - lon2), 2))
        #목표 지점과의 거리가 self.margin_radius보다 작으면
        if radius < self.margin_radius:
            #True(참)를 반환한다.
            return True
```

```
            else:
                #False(거짓)를 반환한다.
                return False

    #check_battery()는 드론의 배터리를 확인하는 함수이다.
    def check_battery(self):
        #rospy.loginfo()로써 self.battery_percent(배터리 잔량의 정보)를 출력한다.
        rospy.loginfo("배터리 잔량: %d%%", self.battery_percent)

    #alti_up()은 이륙한 드론을 일정 고도까지 상승하도록 발행하는 함수이다.
    def alti_up(self):
        #현재의 고도가 목표의 고도보다 낮을 경우에
        while self.alti_bar < FLIGHT_ALT:
            #드론이 상승하도록 tw 메시지를 발행한다.
            self.tw.linear.z = LIN_SPD
            self.mb.pub0.publish(self.tw)

    #move_to_target()은 일정 지점에서 다른 일정 지점으로 드론이 이동하도록 명령하는 함수이다.
    def move_to_target(self, lat1, lon1, lat2, lon2):
        #현재 목표 좌표에 도착했는지의 여부를 판별하고 만일 아니라면,
        while self.check_arrived(lat2, lon2) is False:
            #현재의 고도값이 목표 지점의 범위에 있지 않다면
            if self.check_alt() is False:
                #목표 지점보다 높을 경우에 고도를 낮추고
                if self.alti_bar > FLIGHT_ALT:
                    self.tw.linear.z = -0.1
                #목표 지점보다 낮을 경우에 고도를 높인다.
                else:
                    self.tw.linear.z =  0.1
            else:
                self.tw.linear.z = 0.0

            #목표 좌표를 향하는 각도가 일정 범위 안에 들어 온다면,
            if self.check_route(lat2, lon2) is True:
                #앞으로 이동한다.
                self.tw.linear.x = LIN_SPD
                self.mb.pub0.publish(self.tw)

            else:
                #만일, 목표 좌표를 향하는 각도가 일정 범위 안에 들어오지 않는다면,
```

```
                #현재의 좌표를 다시 구하고,
                lat1, lon1 = self.get_gps_now()
                #현재 좌표와 목표 좌표의 각도를 구한 후
                self.bearing_ref = self.get_bearing(lat1, lon1, lat2, lon2)
                #드론의 머리를 목표의 좌표를 향하도록 회전시킨다.
                self.rotate(lat2, lon2, radians(45))

        #드론이 목표의 좌표에 도착했다면, 움직이지 않는다.
        self.tw.linear.x = 0
        self.mb.pub0.publish(self.tw)
        rospy.sleep(2.0)

        #비밥이 정북을 바라보도록 한다. rotate()의 인자로 북쪽의 목표 좌표를 인위적으로 만들어 전달해주어 실행시킨다.
        rot_lati = self.lati_now + 10
        rot_long = self.long_now
        self.rotate(rot_lati, rot_long, radians(0))
            print("목표의 gps 위치({}, {}) {}에 도착했습니다." .format(self.lati_now, self.long_now, self.alti_gps))

    #fly_to_target()은 드론이 이륙하고 머리를 목표 지점으로 돌리며 그곳으로 이동하도록 하는 함수이다.
    def fly_to_target(self, p2_lati_deg, p2_long_deg):
        if not rospy.is_shutdown():
            #이륙한다.
            self.mb.takeoff()
            #카메라가 정면을 바라본다.
            self.mb.cam_control(0)
            #드론을 일정 높이까지 고도를 높인다.
            self.alti_up()

            #현재의 위도와 경도를 지역 변수로 저장한다.
            p1_lati_deg = self.lati_now
            p1_long_deg = self.long_now
                print("{}, {}에서 {}, {}으로 이동합니다".format(p1_lati_deg, p1_long_deg, p2_lati_deg, p2_long_deg))

            #현재의 위도와 경도, 목표의 위도와 경도의 각도를 구한다.
            self.bearing_ref = self.get_bearing(p1_lati_deg, p1_long_deg, p2_lati_deg, p2_long_deg)
            #드론의 머리를 목표를 향하도록 한다.
```

```
            self.rotate(p2_lati_deg, p2_long_deg, radians(45))
            #드론이 목표 지점으로 이동하도록 한다.
            self.move_to_target(p1_lati_deg, p1_long_deg, p2_lati_deg, p2_long_deg)
            #도착한 후 카메라가 45도를 바라보도록 한다.
            self.mb.cam_control(-45)

        else:
            exit()

    #회전하지 않고 일정한 방향을 바라본 채 목표 지점으로 이동하는 코드를 아래와 같이 구현한다.
    def fly_without_rotate(self, tar_lati, tar_long):
        self.mb.takeoff()
        self.alti_up()
        #현재의 좌표, 목표 좌표의 정보를 이용하여 사선으로 이동한다.
        #현재의 좌표를 구한다.
        #ori_lati = self.lati_now
        #ori_long = self.long_now
        #print("현재의 좌표는 {}, {}입니다.".format(ori_lati, ori_long))

        #지구 위에서의 gps 이동은 완전히 직선으로 이루어질 수 없으므로 5번 반복한다.
        for i in range(5):
            #현재의 좌표를 구한다.
            ori_lati = self.lati_now
            ori_long = self.long_now
            #print("현재의 좌표는 {}, {}입니다.".format(ori_lati, ori_long))

            #비밥이 정북을 바라보도록 roatate의 인자값을 조정한다.
            self.rotate(ori_lati, ori_long, radians(0))
                        print("{}, {}에서 {}, {}으로 순찰합니다.".format(ori_lati, ori_long, tar_lati, tar_long))

            #목표 좌표와 현재 좌표의 위도 거리, 경도 거리, 총 거리를 구한다.
            start_point = (ori_lati, ori_long)
            target_point = (tar_lati, tar_long)

            #위도 간의 차이는 la_gap에 저장한다. 경도는 시작점을 기준으로 한다.
            la_gap = haversine((ori_lati, ori_long), (tar_lati, ori_long), unit = 'm')

            #경도 간의 차이는 lo_gap에 저장한다. 위도는 시작점을 기준으로 한다.
            lo_gap = haversine((ori_lati, ori_long), (ori_lati, tar_long), unit = 'm')
```

```
            #총 거리는 dis에 저장한다.
            dis = haversine(start_point, target_point, unit = 'm')
            #위도의 거리(la_gap), 경도의 거리(lo_gap), 총 거리(dis)를 구한다.

              #print("위도 간 차이는 {}이고, 경도간 차이는 {}이며, 거리는 {}입니다.".format(la_gap, lo_gap, dis))
            #위도 거리, 경도거리, 총거리의 비율을 고려하여 x속도, y속도를 비례적으로 적용한다.
            xdeno = la_gap + lo_gap
            xdeno = abs(xdeno)

            #목표 지점의 위도가 더 높으면 음수를 주고, 아니면 양수를 준다.
            if tar_lati >= ori_lati:
                xnume = la_gap
            elif tar_lati < ori_lati:
                xnume = -la_gap
            else:
                pass
            xspeed = xnume/xdeno * LIN_SPD

            ydeno = la_gap + lo_gap
            ydeno = abs(ydeno)

             #위도와 달리, 목표 지점의 경도가 더 높으면 양수를 주고, 아니면 음수를 준다. 왜냐하면, 비밥은 -y값이 오른쪽으로 향하기 때문이다.
            if tar_long >= ori_long:
                ynume = -lo_gap
            elif tar_long < ori_long:
                ynume = lo_gap
            else:
                pass

            yspeed = ynume/ydeno * LIN_SPD
            #지구는 완전한 평면도 아니고, 완전한 구면도 아니므로 절반만 이동한 후, 위의 과정을 되풀이 하여 정확한 이동이 가능하도록 한다.
            self.mb.move_xy(dis, 0.5, xspeed, yspeed)

        #print("이동을 완료했습니다.")
        #현재의 좌표를 다시 구한다.
        #ori_lati = self.lati_now
```

```
        #ori_long = self.long_now
        #print("현재의 좌표는 {}, {}입니다.".format(ori_lati, ori_long))
```

아) 모듈 파일:MoveBB2_3.py

```
#! /usr/bin/env python
#-*- coding: utf-8 -*-

import rospy
from geometry_msgs.msg import Twist
from std_msgs.msg import Empty
from math import radians, degrees, pi, sqrt
from bb2_pkg.msg import Pos_XYZ_th

#드론의 전후진 운동을 0.200m/s로 설정한다.
LIN_SPD = 0.200
#드론의 전후진 운동의 최솟값을 0.100m/s로 설정한다.
LIN_MINSPD = 0.100
#드론의 회전 속도값을 0.50rad/s로 설정한다.
ANG_SPD = 0.50

#드론의 다양한 비행을 class MoveBB2에서 정의하고 이를 활용한다.
class MoveBB2:
    def __init__(self):
        #드론의 위치가 어디인지에 대한 정보를 구독한다.
        rospy.Subscriber('/bb2_pose_odom', Pos_XYZ_th, self.get_pos_xyzth_cb)
        #드론의 전후진, 회전 등에 관한 퍼블리셔를 초기화한다.
        self.pub0 = rospy.Publisher('/bebop/cmd_vel', Twist, queue_size = 1)
        #드론의 이륙에 관한 퍼블리셔를 초기화한다.
        self.pub1 = rospy.Publisher('/bebop/takeoff', Empty, queue_size = 1)
        #드론의 착륙에 관한 퍼블리셔를 초기화한다.
        self.pub2 = rospy.Publisher('/bebop/land',    Empty, queue_size = 1)
        #드론의 비상 착륙에 관한 퍼블리셔를 초기화한다.
        self.pub3 = rospy.Publisher('/bebop/reset',   Empty, queue_size = 1)
        #드론의 카메라를 컨트롤하는 퍼블리셔를 초기화한다.
        self.pub4 = rospy.Publisher('/bebop/camera_control', Twist, queue_size = 1)

        #Empty() 메시지의 객체 self.empth_msg를 만든다.
        self.empty_msg = Empty()
        #Pos_XYZ_th() 메시지의 객체를 self.xyzth_now, self.xyzth_org 두 개 만든다.
        self.xyzth_now = self.xyzth_org = Pos_XYZ_th()
```

```
        rospy.sleep(3.0)

    #get_pos_xyzth_cb()는 '/bb2_pose_odom'을 구독할 때 자동으로 실행되는 콜백 함수이다.
    def get_pos_xyzth_cb(self, msg):
        #드론의 '/bb2_pose_odom' 토픽을 구독할 때 그 메시지를 self.xyzth_now에 저장한다.
        self.xyzth_now = msg

    #update_org()는 self.xyzth_now를 self.xyzth_org에 치환한다. self.xyzth_now는 콜백 함수 get_pos_xyzth_cb()에서 생성되는 드론의 위치 정보를 담고 있는 변수이다.
    def update_org(self):
        self.xyzth_org = self.xyzth_now

    #elapsed_dist()는 드론이 업데이트한(update_org()) 위치로부터 현재의 위치까지 얼마나 떨어졌는지를 계산하는 함수이다.
    def elapsed_dist(self):
        #삼각함수를 통해서 거리를 측정한다.
        #self.xyzth_now는 콜백 함수 'get_pos_xyzth_cb()'를 통해서 갱신되는 위치 정보이고, self.xyzth_org는 update_org()를 통해서 갱신되는 드론의 위치 정보이다.
        #계산된 이동 정보를 반환한다.
        return sqrt(pow((self.xyzth_now.x - self.xyzth_org.x), 2) + pow((self.xyzth_now.y - self.xyzth_org.y), 2))

    #elapsed_angle()은 드론이 업데이트한(update_org()) 각도로부터 현재의 각도까지 얼마나 벌어졌는가를 계산하는 함수이다.
    def elapsed_angle(self):
        #self.xyzth_now.th는 콜백함수 'get_pos_xyzth_cb()'를 통해서 갱신되는 각도 값이고, self.xyzth_org.th는 update_org()를 통해서 갱신되는 드론의 각도 값이다.
        #계산된 각도의 차이를 반환한다.
        return abs(self.xyzth_now.th - self.xyzth_org.th)

    #elapsed_height()는 드론이 업데이트한(update_org())한 높이로부터 현재의 높이까지 얼마나 벌어졌는가를 계산하는 함수이다.
    def elapsed_height(self):
        #self.xyzth_now.z는 콜백 함수 'get_pos_xyzth_cv()'를 통해서 갱신되는 높이 값이고, self.xyzth_org.z()는 update_org()를 통해서 갱신되는 드론의 높이 값이다.
        return abs(self.xyzth_now.z - self.xyzth_org.z)

    #cam_control()은 드론의 카메라를 일정 각도로 꺽도록 하는 함수이다.
    def cam_control(self, angle):
```

```
        #Twist() 메시지의 객체를 초기화한다.
        tw = Twist()
        #카메라의 각도에 인자로 넘어온 값을 치환한다.
        tw.angular.y = angle
        #카메라의 각도를 일정하게 꺾도록 트위스트 메시지를 발행한다.
        self.pub4.publish(tw)

    #move_x()는 드론의 전후진을 가능하게 하는 함수이다. 거리(distance), 오차 범위(tolerance)를 매개변수로 취한다.
    def move_x(self, distance, tolerance):
        #트위스트(Twist()) 객체를 tw로 만든다.
        tw = Twist()
        #일단, 드론의 움직임에 관한 모든 값들을 0.0으로 초기화한다.
        tw.linear.x = 0.0
        tw.linear.y = 0.0
        tw.linear.z = 0.0
        tw.angular.z = 0.0

        #거리가 0보다 크다면
        if distance >= 0:
            #전진 속도를 LIN_SPD로 주어 전진한다.
            tw.linear.x =  LIN_SPD
        #거리가 0보다 작다면,
        else:
            #드론을 후진시킨다. 즉, linear.x에 음수를 준다.
            tw.linear.x = -LIN_SPD
        #현재 시점의 위치로 시작 지점을 갱신한다.
        self.update_org()
        #드론의 이동 거리가 인자로 넘어온 거리 및 오차 범위보다 작으면
        while self.elapsed_dist() < abs(distance) - abs(distance) * tolerance:
            #트위스트 메시지를 발행하여 전후진으로 움직인다.
            self.pub0.publish(tw)

        #전후진을 마쳤다면, 다시 멈추고 대기한다.
        tw.linear.x = 0;     self.pub0.publish(tw) # stop move
        rospy.sleep(2.0)

    #move_y()는 드론을 좌우 운동하도록 하는 함수이다. 매개변수로 거리(distance), 오차 범위(tolerance)를 취한다.
    def move_y(self, distance, tolerance):
        #트위스트 메시지(Twist())의 객체를 만든다.
```

```
        tw = Twist()
        #일단, 드론의 움직임에 대한 트위스트 메시지를 모두 0.0으로 초기화한다.
        tw.linear.x = 0.0
        tw.linear.y = 0.0
        tw.linear.z = 0.0
        tw.angular.z = 0.0
        #인자로 넘어온 거리(distance)가 0보다 크면,
        if distance >= 0:
            #왼쪽으로 이동한다. 왼쪽이 양수 값을 취한다.
            tw.linear.y =  LIN_SPD
        #인자로 넘어온 거리(distance)가 0보다 작으면
        else:
            #오른쪽으로 이동한다. 오른쪽이 음수 값을 취한다.
            tw.linear.y = -LIN_SPD
        #현재 시점으로 시작 위치를 갱신한다.
        self.update_org()
        #시작 위치로부터의 거리가 인자로 넘어온 거리 및 그 오차 범위보다 작다면,
        while self.elapsed_dist() < abs(distance) - abs(distance) * tolerance:
            #앞서 처리된 트위스트 메시지를 발행하여 일정하게 드론이 좌우 운동을 하도록
 한다.
            self.pub0.publish(tw)
        #이동을 완료했다면, 멈추고 일정 시간동안 대기한다.
        tw.linear.y = 0;      self.pub0.publish(tw)
        rospy.sleep(2.0)

    #move_xy()는 드론이 대각선으로 움직이도록 하는 함수이다. 매개변수로 거리
(distance), 오차 범위(tolerance), 전후진의 속도(xspeed)를 취한다.
    def move_xy(self, distance, tolerance, xspeed, yspeed):
        #트위스트 메시지의 객체를 만든다.
        tw = Twist()
        #인자들을 모두 float형으로 변환한다. 이렇게 하지 않을 경우, 인자로 정수가 넘어
올 때 나눗셈 부분에서 오류를 범한다.
        distance = float(distance)
        xspeed = float(xspeed)
        yspeed = float(yspeed)

        #전후진 운동의 값이 양수이면 앞으로 이동하고 음수이면 뒤로 이동한다.
          #xspeed는 거리에 비례하게 값이 정해지도록 한다. 즉, 거리가 짧을수록
 LIN_MINSPD에 가까운 속도를, 거리가 길수록 인자로 전달받은 xspeed에 가까운 속도를 내
도록 한다.
        if xspeed > 0:
```

```
            xspeed = ((distance/(distance+1))*xspeed) + LIN_MINSPD
        elif xspeed < 0:
            xspeed = ((distance/(distance+1))*xspeed) - LIN_MINSPD
        else:
            xspeed = 0
        #print("x속도는 {}입니다.".format(xspeed))
        #좌우 운동의 값이 양수이면 왼쪽으로 이동하고 음수이면 오른쪽으로 이동한다.
        #xspeed와 마찬가지로, yspeed도 거리에 비례하게 값이 정해지도록 한다.
        if yspeed > 0:
            yspeed = ((distance/(distance+1))*yspeed) + LIN_MINSPD
        elif yspeed < 0:
            yspeed = ((distance/(distance+1))*yspeed) - LIN_MINSPD
        else:
            yspeed = 0
        #print("y속도는 {}입니다.".format(yspeed))
        tw.linear.x = xspeed
        tw.linear.y = yspeed
        #시작 위치를 현재의 위치로 갱신한다.
        self.update_org()
        #시작 위치로부터의 거리가 인자로 전달받은 거리 및 오차 범위보다 작은 동안,
        while self.elapsed_dist() < abs(distance) - abs(distance)*tolerance:
            #트위스트 메시지를 드론에 발행한다.
            self.pub0.publish(tw)
        #print("이동을 완료했습니다.")

    #move_z()는 드론의 일정 높이(고도)로 상하 이동하도록 하는 함수이다.
    def move_z(self, height, tolerance):
        #트위스트 메시지의 객체를 만든다.
        tw = Twist()
        #일단, 전후진 · 좌우 등 드론의 이동에 관한 메시지의 값들을 0.0으로 초기화한다.
        tw.linear.x = 0.0
        tw.linear.y = 0.0
        tw.linear.z = 0.0
        tw.angular.z = 0.0

        #높이가 0보다 크다면 양수를 준다. 이는 드론이 더 높은 고도로 올라간다는 의미이
다. 0보다 작다면 음수를 주고 이는 드론이 더 낮은 고도로 하강한다는 의미이다.
        if height >= 0:
            tw.linear.z =  LIN_SPD
        else:
            tw.linear.z = -LIN_SPD
```

```
        #드론의 시작 위치를 현재 위치로 갱신한다.
        self.update_org()
        #시작 위치로부터의 거리가 인자로 넘어온 높이값 및 오차 범위보다 작다면,
        while self.elapsed_height() < abs(height) - abs(height) * tolerance:
            #트위스트 메시지를 발행하여 드론이 일정 고도에 도달하도록 한다.
            self.pub0.publish(tw)

        #드론이 일정 고도에 도달했다면 멈추고 잠시 대기한다.
        tw.linear.z =  0;  self.pub0.publish(tw) # stop move
        rospy.sleep(2.0)

    #rotate()는 드론의 머리가 일정 방향을 바라보도록 회전시키는 함수이다. 매개변수로 각도, 오차 범위를 취한다.
    def rotate(self, deg, tolerance):
        #트위스트 매시지의 개체를 만든다.
        tw = Twist()
        #인자로 넘어온 각도를 라디안(radian)으로 변환한다.
        angle = radians(deg)
        #트위스트의 이동에 관한 메시지을 0.0으로 초기화한다.
        tw.linear.x = 0.0
        tw.linear.y = 0.0
        tw.linear.z = 0.0
        tw.angular.z = 0.0

        #각도(라디안)가 0보다 크다면, 왼쪽으로 회전한다. 왼쪽 방향의 회전(즉, 시계 운동 방향의 반대)은 양수의 운동값을 취한다.
        if angle >= 0:
            tw.angular.z =  ANG_SPD
        #각도(라디안)가 0보다 작다면, 오른쪽으로 회전한다. 오른쪽 방향의 회전(즉, 시계 운동의 방향)은 음수의 운동값을 취한다.
        else:
            tw.angular.z = -ANG_SPD
        #드론의 현재 위치를 시작 위치로 갱신한다.
        self.update_org()
        #시작 위치로부터의 거리가 인자로 넘어온 각도 및 오차 범위보다 작은 동안
        while self.elapsed_angle() < abs(angle) - abs(angle) * tolerance:
            #드론의 머리가 일정 방향을 가리키도록 트위스트 메시지를 발행한다.
            self.pub0.publish(tw)
        #드론의 머리를 일정 각도까지 꺾었다면, 멈추고 대기한다.
        tw.angular.z =  0;  self.pub0.publish(tw) # stop move
```

```
        rospy.sleep(2.5)

    #takeoff()는 드론이 이륙하도록 하는 함수이다.
    def takeoff(self):
        self.pub1.publish(self.empty_msg)
        #print("이륙합니다.")

    #landing()은 드론이 착륙하도록 하는 함수이다.
    def landing(self):
        self.pub2.publish(self.empty_msg)
        #print("착륙합니다")

    #emergency()은 드론이 비상 착류하도록 하는 함수이다.
    def emergency(self):
        self.pub3.publish(self.empty_msg)
        #print("emergency")
```

6. 화재 인식을 위한 xml 파일 만들기

화재 인식을 위한 학습 xml파일을 만들기 위하여 본 팀은 Cascade Trainger GUI(Version 3.3.1)을 썼다. 그 과정은 대략 다음과 같다.

1. 학습시키고자 하는 화재 영상을 찍는다.
2. 화재 영상에서 사진을 추출한다.
3. 화재의 사진과 그와 대비되는 사진을 분류한다.
4. 캐스케이드 프로그램에 사진이 포함된 폴더를 가리키고, 사진의 크기 · 사진의 수 · 개채의 특징 등 여러 매개변수들을 변동시켜가며 학습시킨다.
5. 캐스케이드 프로그램의 학습 결과로 나온 xml 파일을 실제 영상에 적용시켜 본다.

그 결과는 별책부록으로 제시한다.

7. 실험 결과

2021년 충남인력개발원의 ROS Fire_Detection_Drone 팀은 8월 16일부터 9월 2일까지 수십 차례에 걸쳐 코드를 테스트하였다. 그 중 유의미한 결과나 나온 일부 내용을 아래와 같이 정리한다.

가) 2021년 8월 30일

2021년 8월 30일 우리팀은 충남인력개발원 본관 앞 운동장에서 화재 감지 드론의 코드를 검증하였다.

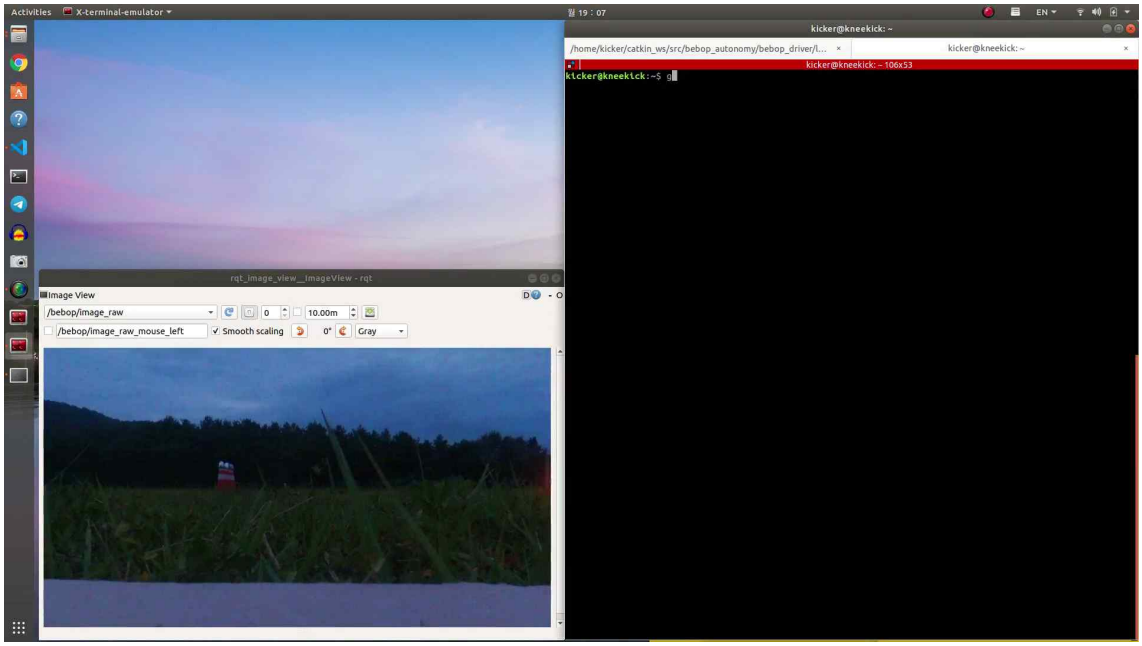


그림 3

위 그림(3)은 rqt 화면, 실행 터미널을 각각 보여준다. rqt 화면에서는 드론의 실시간 화면이 나타나고 터미널에서는 실행 명령 및 그 결과 들을 볼 수 있다. 드론이 목표 지점에 도달했다면 새로운 화면 창이 뜨고, 화재 감지 코드가 활성화될 것이다.

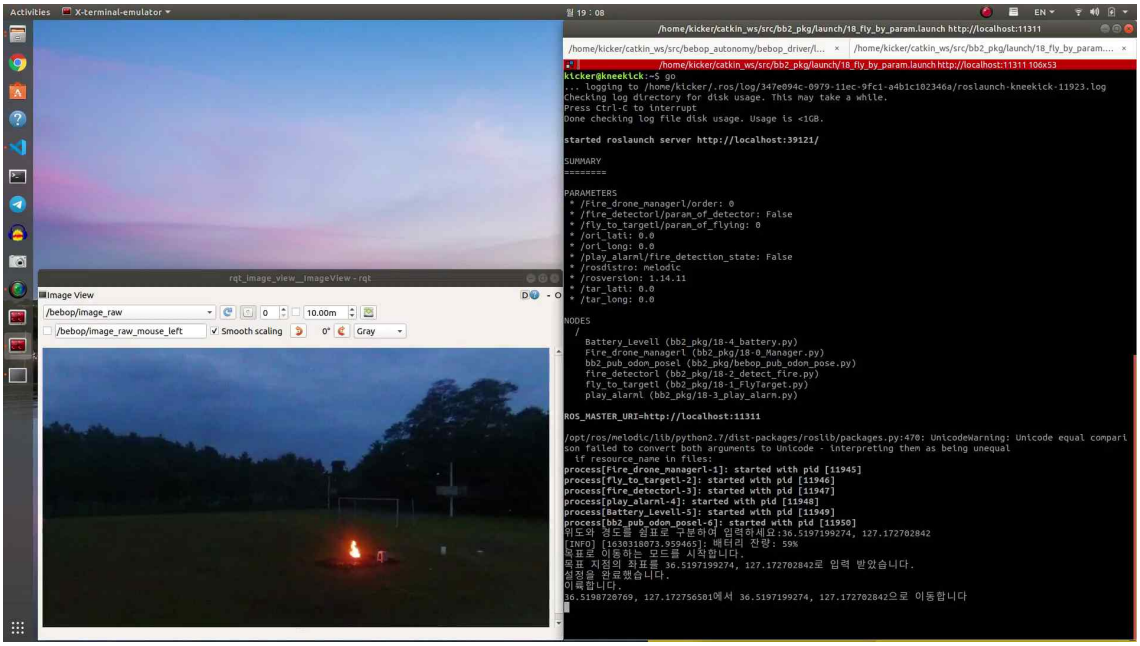


그림 4

우리 팀이 짠 코드는 일정 GPS 지점까지 도달할 때까지 화재 감지를 작동하지 않는다. 그 이유는 드론의 목적이 일정 지점의 일정 범위에서 화재가 발생했는지의 여부를 수색하는 것이기 때문이다. 그러므로 위의 그림(4)에서 보는 것과 같이 목표 지점에 도달하기 rqt에서 보이는 영상은 화재 탐지의 대상이 되지 않는다. 사용자가 입력한 GPS 지점에 도달하면 화재 감식 코드가 활성화되고 새로운 프레임 창이 뜬다.

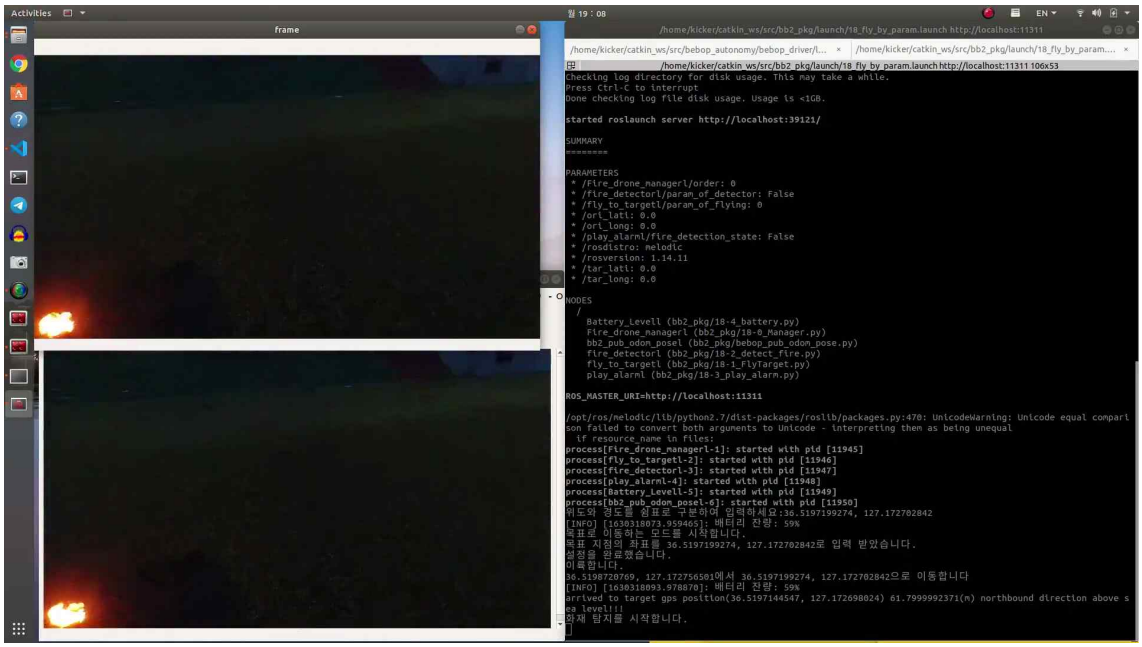


그림 5

드론이 사용자가 입력한 GPS 지점에 도착하면 위의 그림(5)에서 보는 것과 같이 새로운 프레임이 생기고 화재 감식 코드를 활성화한다. 만일 화재가 감식되면 화재로 판단된 객체에 파란색 박스가 둘러쳐지고 그것을 화면의 중앙에 맞추는 비행 코드가 작동한다.

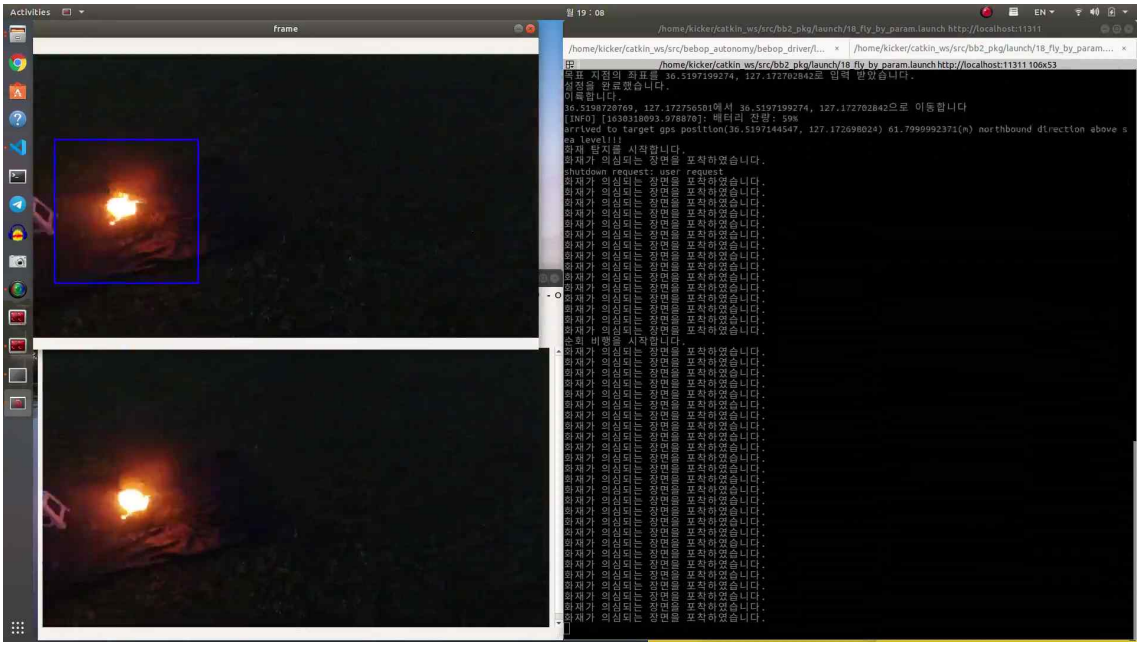


그림 6

그림(6)에서는 드론 코드가 화재를 인식하고 그것에 파란색 박스를 친 것을 볼 수 있고, 터미널 창에서는 “화재로 의심되는 장면을 포착하였습니다.”라는 메시지가 뜬다. 즉, 입력된 영상에서 화재를 감지한 것이다. 그런데 화재가 아닌 의자도 화재로 오인식함도 엿보인다. 이처럼 이번 프로젝트에서는 화재 인식의 정확도가 매우 중요하다.

만일, 실제 화재가 아닌 객체를 화재로 인식할 가능성이 있다. 이를 방지하기 위하여 코드에서는 화재의 객체가 화면의 중앙에 올 때까지 화재 인식을 10초 이내로 끊김없이 이어지도록 해야 한다는 조건을 만들었다.

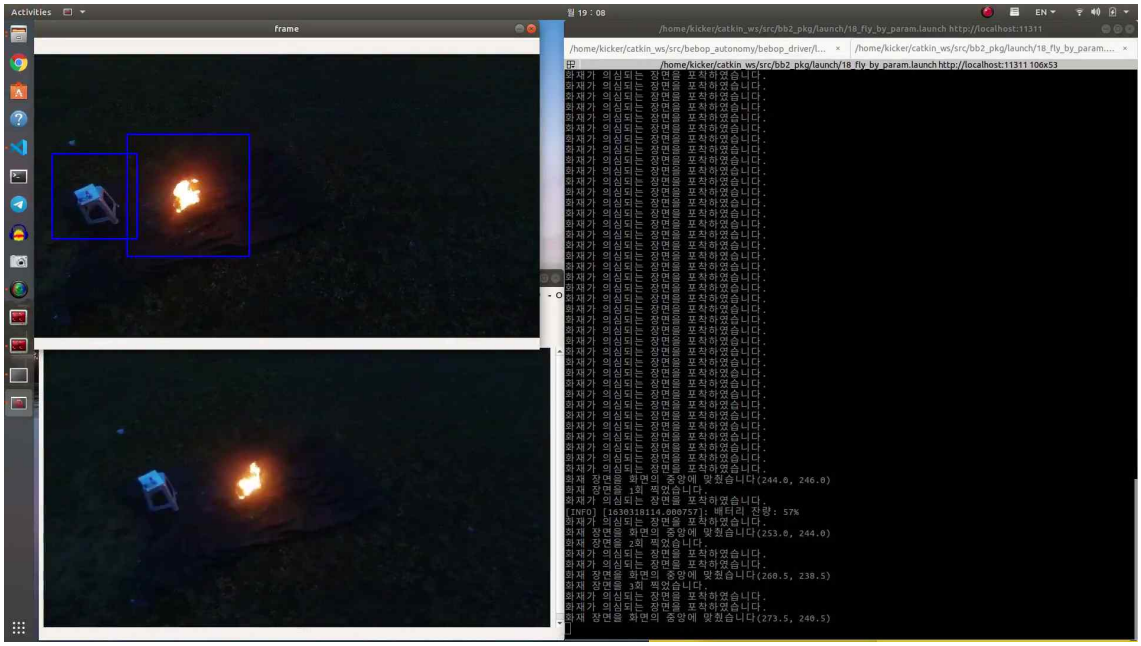


그림 7

화재로 감지된 객체가 화면의 중앙 범위에 맞추어졌을 때, 코드에서는 해당 영상의 이미지를 3회 연속해서 저장한다. 그런 다음 사용자에게 어느 (GPS상의) 위치에서 화재가 발생했는지를 알리고 시작 지점으로 되돌아 간다. 사용자에게는 터미널 창에서 표준 출력함과 동시에 문자 메시지로 알리는 기능이 코드에 추가되어 있다.

화재 감식의 임무를 끝냈다면, 드론이 다시 시작지점으로 되돌아 가도록 한다. 따라서 화재 감식을 할 때 보여지는 프레임도 사라지게 된다.

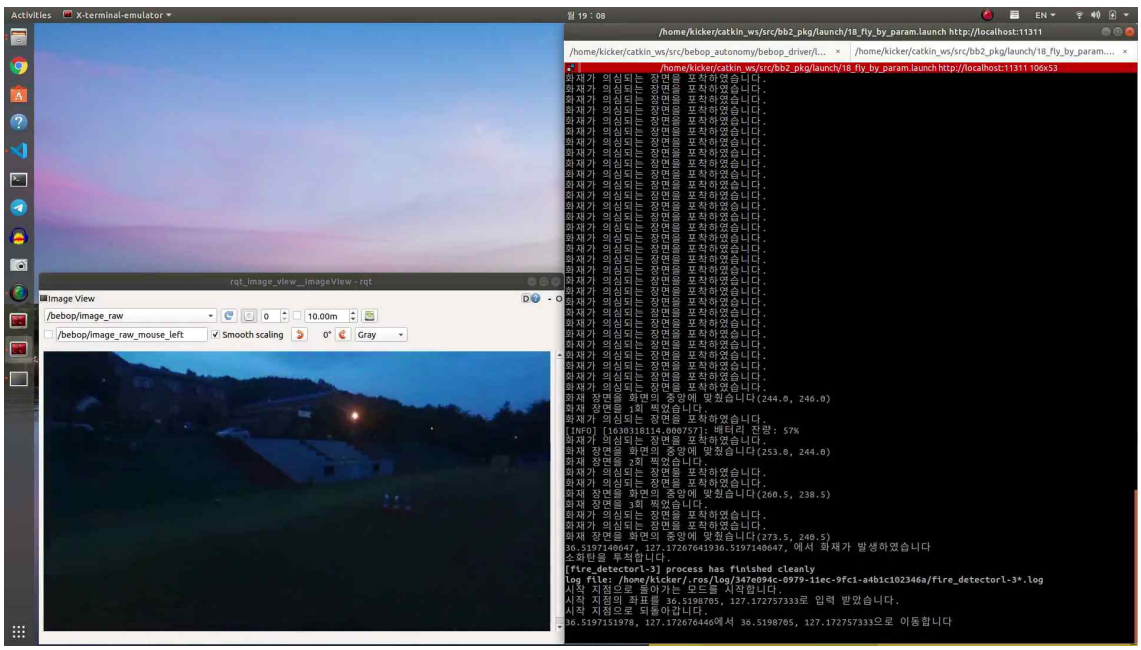


그림 8

8. 참고자료

강욱. “공공임무용 드론의 사회적 비용·편익 분석에 대한 연구”. 한국테러학회보, 제11권 제4호, 2018년.

국토교통부(첨단항공과). 드론 활용의 촉진 및 기반조성에 관한 법률 (약칭: 드론법), Pub. L. No. 제16420호 (2019).

김동현·김성렬. “Open_CV를 활용한 위험 상황 인식에 관안 연구”. Journal of the KIECS, 2021년 2월 16일.

김별·황광일. “딥러닝 기반의 연기 확산거리 예측을 위한 알고리즘 개발 기초연구”. Journal of the Korean, 2021년.

노예진. “’특파원보고 세계는지금’기후변화로 인한 섭씨 50도의 폭염...최악의 산불”, 2021년 8월 14일.

문준호·최혁두·박남훈·김종희·박용운·김은태. “데이터베이스 기반 GPS 위치 보정 시스템”. Jouranl of Korea Robotics Society, 2012년.

배용민. “[119기고] 소방 드론의 활용”. 소방방재신문, 2021년 4월 7일. https://www.fpn119.co.kr/155045.

서울소방재난본부. “재난현장에서 드론의 운용 활성화를 위한 방안 연구”. 제29회 119소방 정책 컨퍼런스, 2017년 9월.

소방안전플러스 편집실. “소방안전의 파수꾼 드론[drone]”. 소방안전플러스, 2021년 4월 7일. http://webzine.kfsa.or.kr/user/0008/nd46782.do#.

소방청, 편집자. 2020년도 화재통계연감, 2021.

신열우·박진호. “경험적 접근법을 활용한 재난현장에서의 소방드론 임무수행 효율성 분석”. 한국화재소방학회, 한국화재소방학회논문지 Vol.34 No.5, 2020년.

조성완. “소방공무원의 고령화에 따른 자기효능감, 팀워크가 현장대응역량에 미치는 영향과 제도 개선방안에 대한 연구”. 서울시립대학교, 2016.

하강훈, 김재호&최재욱. “소방분야의 드론 활용방안 연구 경향 분석”. 한국산학기술학회논문지 제22권 제4호, 2021년. https://www.koreascience.or.kr/article/JAKO202113855736456.pdf.

“화마가 덮친 남유럽, 통제불능 기후변화”. 세계는지금. KBS, 2018년 8월 14일.