# Development and testing of methods for drones control

**Paolo Leopardi**
paolo.leopardi96@gmail.com

Last update: November 6, 2023

## Contents

## Acronyms

**CPP** Coverage Path Planning

**GCS** Ground Control Station

**QGC** QGroundControl

**ROS** Robot Operating System

**SITL** Software-In-The-Loop

**TSP** Travelling Salesman Problem

**UAV** Unmanned Aerial Vehicle

**WSL** Windows Subsystem for Linux

# 1 Introduction

The project is officially called (italian) *Sviluppo e sperimentazione di metodologie di controllo per droni* and it lies in the field of agricultural robots. The aim of these months will be the implementation of a real drone which is able to move in a area and take informations autonomously through some device mounted on the robot (thermal imager). The project can be splitted in two main parallel directions:

- Navigation (path planner, control schemes, etc.)
- Implementation (drone construction, autopilot selection, etc.)

# 2 Navigation

To successfully cover the area of interest the quadcopter has to navigate following a certain logic and take into account different factors such as the shape and the dimension of the area, presence of obstacles, vehicle used and so on. There is a class of algorithm called Coverage Path Planning (CPP) which is well suited for this aim.

## 2.1 Coverage Path Planning

Given an area of interest the CPP problem consist of planning a path which covers the entire target environment considering the vehicle's motion restriction and sensor's characteristics, while avoiding passing over obstacles [1]. These algorithms can be classified into two main categories: offline and online [2]. Offline algorithms need a previous knowledge of the search area, online algorithm instead are based on real-time data acquisition.

Another classification is based on the decomposition method thaat can be classified in:

- Cellular decomposition
  - Exact decomposition
  - Approximate decomposition
- No decomposition

Cellular decomposition methods are based in dividing the surface into cells: in the exact decomposition the workspace is splitted in sub-areas whose re-union exactly occuped the target area [1]. In the approximate decomposition the area is usally divided using a grid where the size of the squares is typically determined for example by the footprint of the camera mounted on the robot. In no decomposition techniques, as the name suggest, isn't applied any type of decomposition. Taking in to account the aim of the project, i.e. the Unmanned Aerial Vehicle (UAV) has to collect data in different positions of an area, the best solution is the approximate decomposition technique beacuse we don't need to cover every centimetre of the area (like an autonomous lawn mower), we need to determine the amount of waypoints that guarantees an exhaustive data collection compared to the target area.

# 3 Implementation

The implementation step is the physical construction of the UAV, this involves the selection of all the elements, both hardware (e.g. platform and its components) and software (e.g. autopilot flight stack).

## 3.1 Autopilot selection

Autopilot selection is made by evaluating possible pros and cons which every autopilot fligh stack brings with it. Three possible solution were evaluated:

1. INAV [3]
2. PX4 [4]
3. Agilicious [5]

There are a lot of reason which can determine the choice of one solution instead of another, a preliminary evaluation is made considering the informations available on the web (official documentation and other sources). These parameters have been accounted:

- configuration
- missions definition
- future developments

*Configuration* denotes the level of complexity needed to configure flight controller for the first flight, *missions definition* takes into account how to define missions, and *future development* indicates compatibility with other framework, software and so on.

INAV's configuration seems easy as PX4, the main difference is the guide: for INAV you can follow some videos on Youtube at this link, for PX4 it's necessary to follow sections from *Basic Assembly* to *Flying* in the official documentation. Agilicious doesn't have a section related to the configuration steps for the first real flight like the above mentioned.

INAV provide a Ground Control Station (GCS) which is capable of define only waypoints which the UAV has to visit, as shown for example here. PX4 typically use QGroundControl (QGC) as GCS[1], here different missions can be defined and it is worth to note that there is also survey missions which seems particularly suited with the aim of this project. Agilicious doesn't not provide a GCS for missions definition, but it has a module called `reference` which implements different ways of generating reference trajectories.

I wasn't able to find any documentation regarding interfacing between INAV and Robot Operating System (ROS), PX4 has a subsection dedicated to ROS communication with PX4. In addiction PX4 has a MATLAB package called UAV Toolbox Support Package for PX4 Autopilots [6]. Agilicious has a very good structure for future developments beacause you can change controller or estimator by simply modify a `yaml` file. It's not provided a way to integrate GPS measurements. An interface for ROS called `agiros` is provided. Both PX4 and Agilicious docs propose a simulator.

In conclusion the better idea should be to try the autopilot in this order: PX4, Agilicious, INAV.

[1]QGC supports only PX4 and Ardupilot

## 3.2 PX4 configuration

Before first flight PX4 Autopilot needs some steps to follow to configure the autopilot, this one are documented in PX4 documentation's section called Standard Configuration. The procedure is quite straightforward but some problems may arise during these steps.

### Troubleshooting

#### Firmware version

QGC provides au automatic way to flash the latest firmware[2], however all version 13 express same problem with our specific hardware. More specifically the problem is related to the Wi-Fi module because with the firmware version `v1.13.x` the autopilot is unable to connect with QGC. So I found that version `v1.12.3`[3] fixes this problem.

#### Autotune

Having downgraded to the version `v1.12.3` determined the impossibility to use the autotune procedure because this is available from `v1.13.0`.

## 3.3 Optitrack configuration

After some outside experiments (in which human pilot successfully drove the quadcopter) we decided to take the next flight test in an indoor scenario; this beacause an indoor environment is safer if compared to the outdoor one in terms of damaged caused by the drone's crashing.

Before flying, the communication between Optitrack and flight controller needs to be configured, we can think the Optritrack as the indoor counterpart of the GPS. To configure the Optritrack with PX4 there also a dedicated section named Using Vision or Motion Capture Systems for Position Estimation, this one provides all the necessary steps to configure the communication. Please note that there is also a dedicated subsection for Optitrack system.

### Troubleshooting

#### Parameters

Having used an older firmware version, some parameters[4] have been replaced with others; these ones are listed in the table below. The first column shows the actual name of the parameters the second columns shows the counterpart on the firmware version used in this project.

| PX4 docs naming | v1.12.3 naming |
|---|---|
| EKF2_EV_CTRL | EKF2_AID_MASK |
| EKF2_HGT_REF | EKF2_HGT_MODE |
| EKF2_GPS_CTRL | EKF2_AID_MASK |

Another set of parameters are the ones used for the preflight check, Disabling these prevents the drone from checking the correct operation of the corresponding sensors:

- `SYS_HAS_BARO`

[2] At the time of writing this report, i.e. September 2023, the last stable release is `v1.13.3`.
[3] Firmware releases available here.
[4] Full parameter list here.

- SYS_HAS_GPS
- SYS_HAS_MAG

# 4 Simulation

To test some CPP algorithms is worth to implement an environment which allows to drive a drone autonomously in a safe way, without any risk of collision. PX4 offers different simulators which allow to develop Software-In-The-Loop (SITL) simulation [7]. More in details, in the section named *MAVROS Offboard control example (Python)* [8] there is a useful example on how to setup PX4, Gazebo and MAVROS to run a simulation.

To implement a complete pipeline which allows to develop CPP algorithms, simulate the quadcopter and analyse the result MATLAB has been employed beside the simulator structure which exploit PX4, Gazebo and MAVROS, mentioned above. MATLAB is used to determine the area, develop the CPP algorithms and pass the waypoint to the simulator. After the simulation phase, which is executed in Gazebo environment the results are visualized and analysed in MATLAB again.

MATLAB is executed in Windows environment, while PX4 software stack, Gazebo and of course ROS are executed in Windows Subsystem for Linux (WSL) environment (more specifically in Ubuntu 20.04).

## 4.1 MATLAB code

There are two main files called main.m and WSL_connection.m; the first one deals with the definition of the target area by the user, the execution of the CPP algorithm and consequently the determination of waypoints in space. The second one is devoted to establish the connection with WSL to send the waypoint calculated in the main.m file.

### Main.m code

As mentioned previously the main.m code is responsible for determining the area of interest and the waypoint calculation. The first section allows the user to select an area by specifying the latitude and longitude as shown in 1. After the selection procedure the area is converted in local coordinate expressed in meters. This further step is needed because waypoints will be calculated based on robot's footprint.

```
%% create target area

basemap = "satellite";
geobasemap(basemap)
longitude = 43.116118;
latitude = 12.525492;
[long_Lowlim, longUpLim] = roundNumber_2digits(longitude);
[lat_LowLim, latUpLim] = roundNumber_2digits(latitude);
geoplot([long_Lowlim, longUpLim], [lat_LowLim, latUpLim])
set(gcf, 'WindowState', 'maximized');

target_area = drawpolygon("Color", 'y');

target_area = polyshape(target_area.Position(:,2), target_area.Position(:,1)
    );
close
```

```

% conversion to local coordinates expressed in metres
target_area_verticesMeters = latLonToMeters(target_area);
target_area_meters = polyshape(target_area_verticesMeters(:,2),
    target_area_verticesMeters(:,1));

subplot(1,2,1)
plot(target_area)
title('Target area')
xlabel('Longitude')
ylabel('Latitude')
grid on

subplot(1,2,2)
plot(target_area_meters)
title('Target area (local coordinates in meters)')
xlabel('X (m)')
ylabel('Y (m)')
grid on
```

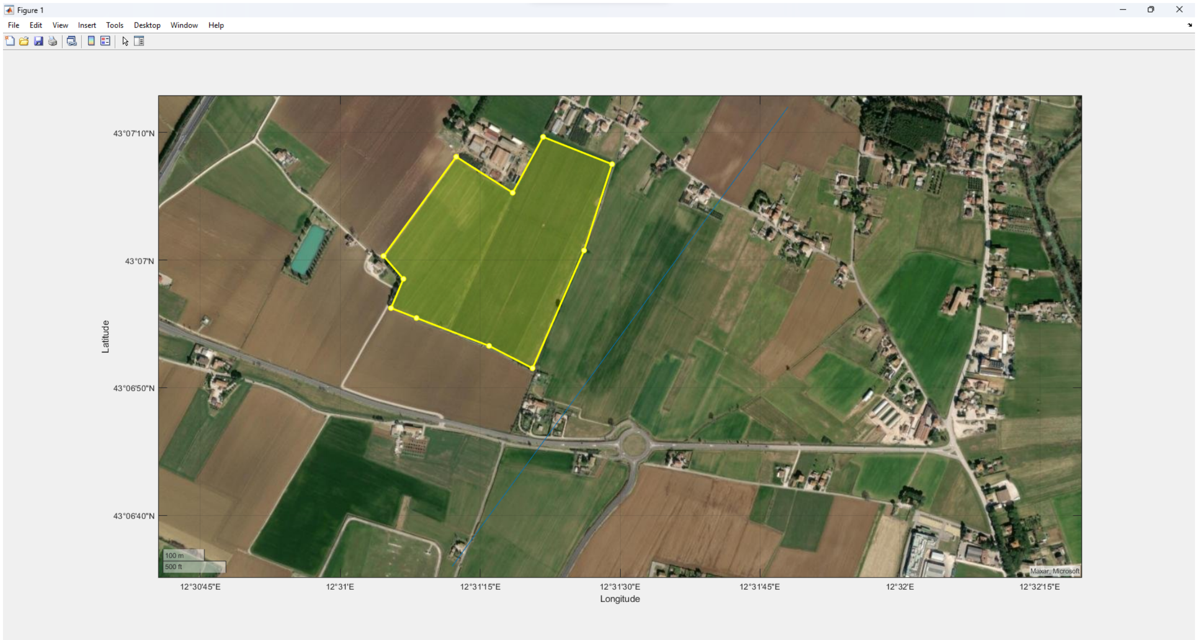


**Figure 1:** Target area selection.

The next section is devoted to the computation of the waypoints related to the selected area.

```
%% compututation of waypoints

 % target_area_meters = polyshape([0 4 4 0], [0 0 3 3]); % only for test

robot_footprint = polyshape([0 50 50 0], [0 0 50 50]);
% specify the footprint of the sensor mounted on the UAV, the algorithm
% will calculate a enough number of waypoints which cover the entire area
% taking into account sensor's footprint

waypoints = calculateWaypoints(target_area_meters, robot_footprint);
```

```
% the function above can be seen as the CPP algorithm if we leave the waypoint in the
% order calculated by the function (of course this waypoints aren't optimized regarding any metrics)

figure
plot(target_area_meters, 'FaceColor', 'g')
hold on
scatter(waypoints(:, 1), waypoints(:, 2), '*r')
title('Target area waypoints')
xlabel('X (m)')
ylabel('Y (m)')
grid on
axis equal

% Set labels to be displayed near the waypoints
labels = 1:size(waypoints, 1);  % Assuming you want to label each waypoint with numbers 1, 2, 3, ...
labelOffsets = [0.15, 0.15];  % Adjust the offsets for label positioning

% Add text labels near the symbols
for i = 1:size(waypoints, 1)
    x = waypoints(i, 1) + labelOffsets(1);
    y = waypoints(i, 2) + labelOffsets(2);
    text(x, y, num2str(labels(i)), 'FontSize', 10);
end
```

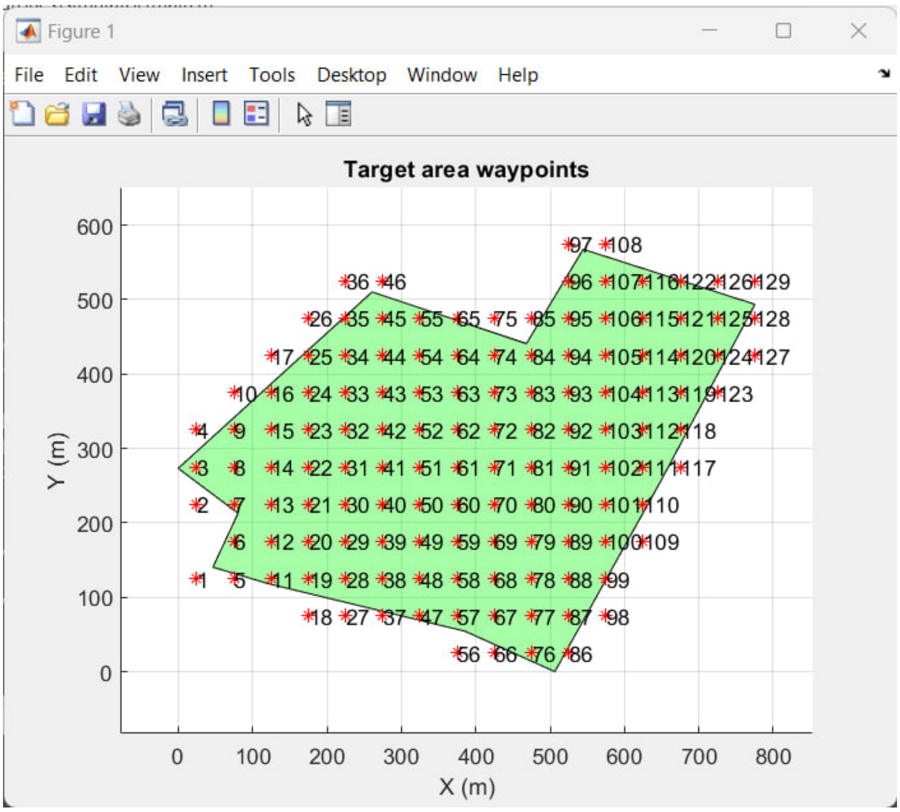


**Figure 2:** Computation of waypoints.

The following part sorts the waypoints by applying the algorithm based on the Travelling Salesman Problem (TSP) [9]. MATLAB solver for a general TSP is available at *Traveling Salesman Problem: Problem-Based* and the function named TSP_waypoints implement this example with minor adjustments. The variable named waypoints_sequence stores the waypoints order, the variable named waypoints_ordered stores the sequence of waypoints computed by the TSP algorithm.

```
%% sort waypoints and add reference altitude

waypoints_sequence = TSP_waypoints(waypoints, target_area_meters) % get
    waypoints sequence
waypoints_ordered = waypoints(waypoints_sequence, :) % sort waypoints by the
     order specified in waypoint_sequence

figure
plot(target_area_meters, 'FaceColor', 'g')
hold on
scatter(waypoints(:, 1), waypoints(:, 2), '*r')
plot(waypoints_ordered(:,1), waypoints_ordered(:,2), 'Color', 'b')
grid on


% until now the waypoints computed where planar waypoints, for now a
% dummy reference altitude of 1 meter is added
ref_height =  transpose(ones(1, size(waypoints_ordered, 1))*1); %% add
    reference altitude to waypoint
waypoints3D = [waypoints_ordered, ref_height]; % waypoints in 3D space
```

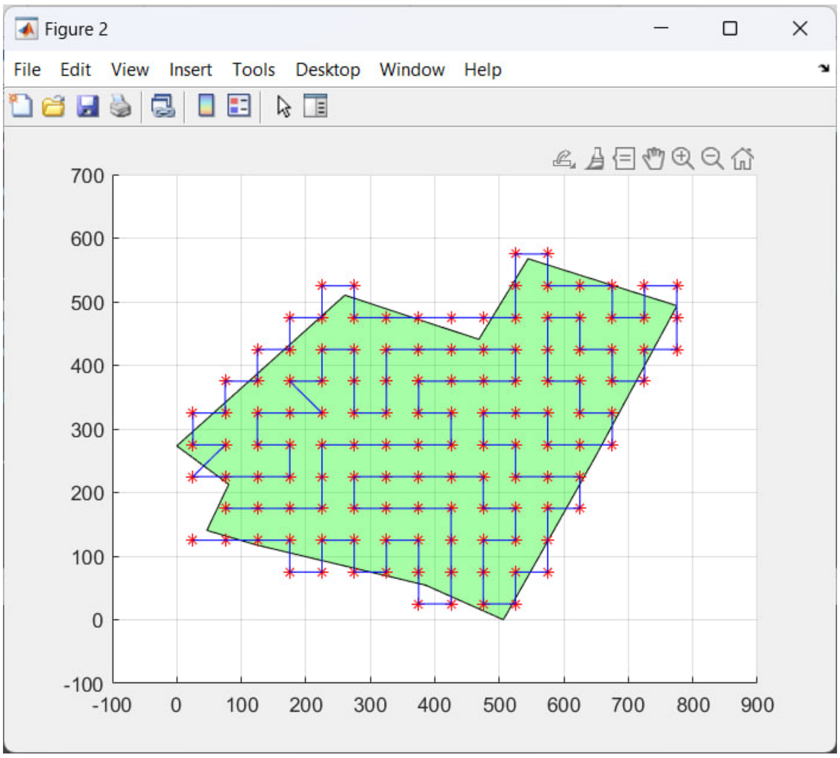


**Figure 3:** Waypoints ordered by the TSP algorithm.

The last section is responsible for the analysis of the data collected in the simulation. An example on how to get the trajectory of the quadcopter from rostopic is provided[5].

## WSL_connection.m code

This script is responsible for the connection with WSL, which means that allows MATLAB to interact with the rostopic executed on WSL, it automatically recovers the ip addresses needed [6].

```
%% connect to WSL

% get ip addresses needed to communicate with WSL

[~, ipAdd_wsl] = system('wsl ip -4 addr show eth0 | findstr "inet"');

ipAdd_wsl = regexp(ipAdd_wsl, '\d+\.\d+\.\d+\.\d+', 'match');
ipAdd_wsl = ipAdd_wsl(1)

[~, ipAdd_windows] = system('ipconfig | findstr /C:"vEthernet (WSL)" /C:"
    IPv4"', '-echo');
ipAdd_windows = regexp(ipAdd_windows, '172.\d+\.\d+\.\d+', 'match')


% wsl ip addres -> $ ifconfig -> eth0, inet
string_ROS_MASTER_URI = strcat('http://', ipAdd_wsl, ':11311')
setenv('ROS_MASTER_URI',string_ROS_MASTER_URI)

% wsl ip on windows -> $ ipconfig -> Scheda Ethernet vEthernet (WSL),
    indirizzo IPV4
setenv('ROS_IP', ipAdd_windows)

rosinit

rostopic list
```

In addition, the second section is responsible of instantiating a new node which publish the waypoint list[7] in the topic named /MATLAB_waypoint.

```
%% publish waypoint

clc

% publisher
waypoint_pub = rospublisher('/MATLAB_waypoint', 'geometry_msgs/PoseArray')


% create message
```

[5]Please note that the file named rosbag_2.bag is only an example and so the data recorded couldn't be related with the current waypoints computed

[6]This code is tested in Windows system, to recover the addresses through Windows terminal and ubuntu shell type the following commands:

- In Ubuntu shell type `$ ifconfig` and look for the address named `iner` under `eth0`. This is the variable named `ipAdd_wsl`.
- In Windows terminal type `$ ip config` and look for `IPv4 address` under `Ethernet Card vEthernet (WSL). This is the variable named ipAdd_Windows`.

[7]The code assumes that there is a matrix variable named waypoint3D, wtih dimensions $N \times 3$, where $N$ represents the number of waypoint and the columns are the $xyz$ coordinates in the space.

```
waypointList_msg = rosmessage('geometry_msgs/PoseArray');

for i = 1:length(waypoint3D)
    waypoint = rosmessage('geometry_msgs/Pose');
    waypoint.Position.X = waypoint3D(i,1);  % Set the x-coordinate
    waypoint.Position.Y = waypoint3D(i,2);  % Set the y-coordinate
    waypoint.Position.Z = waypoint3D(i,3);  % Set the z-coordinate
    waypoint.Orientation.W = 1.0;

    % Add the Pose message to the PoseArray
    waypointList_msg.Poses = [waypointList_msg.Poses; waypoint];

end
```

## 4.2 WSL environment

WSL is used to run the simulations, it exploits the PX4 autopilot with Gazebo and MAVROS. First of all the following components need to be installed:

- PX4 autopitlot folder, downloadable by following the section named *Ubuntu Development Environment*.
- ROS and MAVROS[8]

The idea behind the simulation is to use MAVROS to drive in off-board mode a simulated quadcopter in Gazebo using the waypoints computed in MATLAB. As a result, we need a acros that is capable of subscribing to the topic named /MATLAB_waypoint to recover the waypoint list. By following PX4 documentation sections named *MAVROS Offboard control example (Python)* is easy to understand how to develop a new ROS package[9]. The file implemented to create the rosnode is called waypoint_manager.py.

### Waypoint_manager.py

The code should be easily readable by the user, for more details contact the author.

```
#!/usr/bin/env python3

import rospy
from geometry_msgs.msg import PoseStamped, Point, PoseArray
from mavros_msgs.msg import State
from mavros_msgs.srv import CommandBool, CommandBoolRequest, SetMode,
  SetModeRequest
from math import dist


current_state = State()

waypoint_index = 0
waypointList = []
waypointReceived = False # flag to check if waypoint list is received
nextWaypoint = [0, 0, 0]
```

[8]The installation of ROS and MAVROS is not covered in this guide.

[9]catkin_make can be used instead ofcatkin build, for more details take a look at ROS documentation section named *Creating a ROS Package*.

```python



def state_cb(msg):
  global current_state
  current_state = msg


def buildWPArray(data):
  for index in range(len(data.poses)):
  waypointList.append([data.poses[index].position.x, data.poses[index].
 position.y, data.poses[index].position.z])


def getNextWP(currentPosition, threshold):

  global waypoint_index
  global waypointList
  global nextWaypoint

  try:
    currentWaypoint = waypointList[waypoint_index]
    nextWaypoint = currentWaypoint

    if dist(currentPosition, currentWaypoint) < threshold: # compute
 euclidean 3D distance
    waypoint_index += 1
    nextWaypoint = waypointList[waypoint_index]
    rospy.loginfo('Next waypoint: ' + str(nextWaypoint))

  except IndexError:
    pass


return nextWaypoint



def WP_callback(data):

  if waypointReceived:

  targetWP = getNextWP([data.pose.position.x,
  data.pose.position.y,
  data.pose.position.z], threshold=.2)



  # Create a PoseStamped message
  pose_msg = PoseStamped()
  pose_msg.header.stamp = rospy.Time.now()
  pose_msg.pose.position.x = targetWP[0]
  pose_msg.pose.position.y = targetWP[1]
  pose_msg.pose.position.z = targetWP[2]
  pose_msg.pose.orientation.x = 0.0
  pose_msg.pose.orientation.y = 0.0
  pose_msg.pose.orientation.z = 0.0
```

```
  pose_msg.pose.orientation.w = 0.0

  currentWaypoint_pub.publish(pose_msg)

  #else:
  #    rospy.loginfo('Waiting for waypoint')

if __name__ == '__main__':

  try:

  rospy.init_node('waypoint_manager')

  # subscribers
  state_sub = rospy.Subscriber("mavros/state", State, callback = state_cb)
  position_sub = rospy.Subscriber('mavros/local_position/pose', PoseStamped, callback = WP_callback)

  # publisher
  currentWaypoint_pub = rospy.Publisher('/mavros/setpoint_position/local', PoseStamped, queue_size=10)


  rospy.wait_for_service("/mavros/cmd/arming")
  arming_client = rospy.ServiceProxy("mavros/cmd/arming", CommandBool)

  rospy.wait_for_service("/mavros/set_mode")
  set_mode_client = rospy.ServiceProxy("mavros/set_mode", SetMode)

  waypointMessage = rospy.wait_for_message("/MATLAB_waypoint", PoseArray)
  buildWPArray(waypointMessage)
  rospy.loginfo("Waypoint list: " + str(waypointList))
  waypointReceived = True


  # Setpoint publishing MUST be faster than 2Hz
  rate = rospy.Rate(20)

  # Wait for Flight Controller connection
  while(not rospy.is_shutdown() and not current_state.connected):
    rate.sleep()

  offb_set_mode = SetModeRequest()
  offb_set_mode.custom_mode = 'OFFBOARD'

  arm_cmd = CommandBoolRequest()
  arm_cmd.value = True

  last_req = rospy.Time.now()

  while(not rospy.is_shutdown()):
    if(current_state.mode != "OFFBOARD" and (rospy.Time.now() - last_req) > rospy.Duration(5.0)):
      if(set_mode_client.call(offb_set_mode).mode_sent == True):
        rospy.loginfo("OFFBOARD enabled")

      last_req = rospy.Time.now()
```

```

    else:
      if(not current_state.armed and (rospy.Time.now() - last_req) > rospy.Duration(5.0)):
        if(arming_client.call(arm_cmd).success == True):
          rospy.loginfo("Vehicle armed")

        last_req = rospy.Time.now()

    rate.sleep()

  rospy.spin()

except rospy.ROSInterruptException:
  rospy.logwarn("Node Interrupted")
```

## 4.3 Execute simulation

Is recommendable to don't use a `.launch`, launch all the files from different terminal. These are the steps to successfully run the simulation:

1. Run `main.m` file to compute the waypoints
2. On the first terminal run the command `$ roslaunch PX4-Autopilot/launch/mavros_posix_sitl.launch` to launch the PX4 autopoilot and Gazebo environment
3. On the second terminal run the command `$ rosrun offboard_py waypoint_manager.py` to launch the `waypoint_manager` node
4. Run `WSL_connection.m` to send the waypoint list to the `waypoint_manager` node

After this the situation should look like the one depicted in Figure 4.

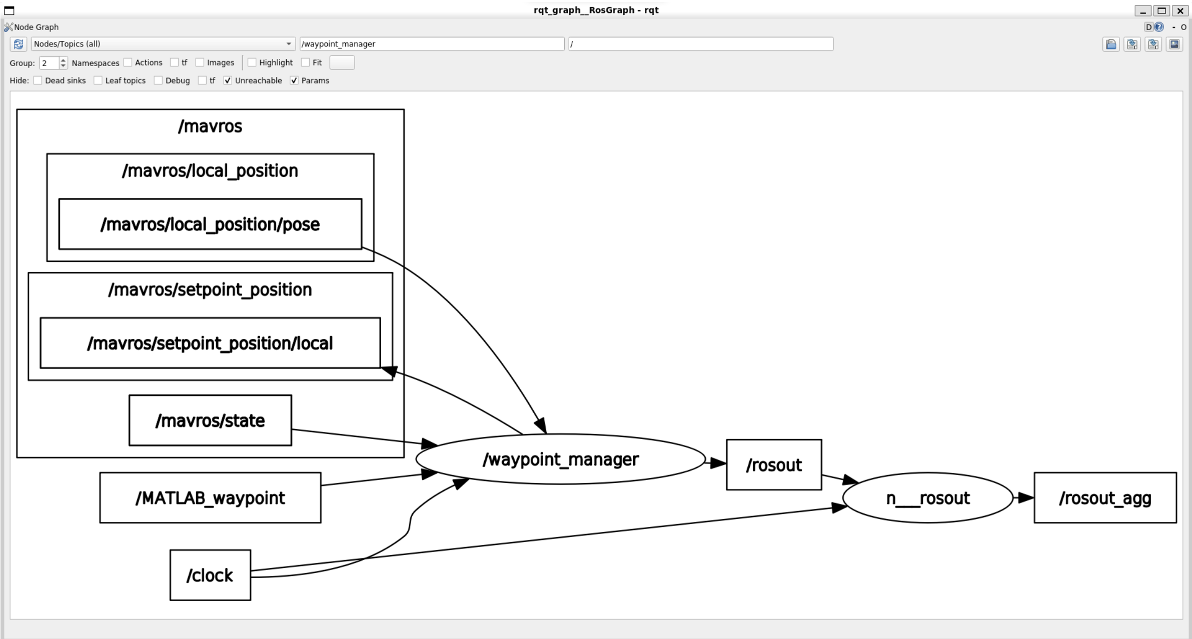


**Figure 4:** Node and topic view focused on `waypoint_manager` node.

## References


[1] Tauã M Cabreira, Lisane B Brisolara, and Ferreira Jr Paulo R. Survey on coverage path planning with unmanned aerial vehicles. *Drones*, 3(1):4, 2019.

[2] Georgios Fevgas, Thomas Lagkas, Vasileios Argyriou, and Panagiotis Sarigiannidis. Coverage path planning methods focusing on energy efficient and cooperative strategies for unmanned aerial vehicles. *Sensors*, 22(3):1235, 2022.

[3] iNavFlight. iNav. Available at: https://github.com/iNavFlight/inav, 2023. Accessed: 12 July 2023.

[4] PX4 Autopilot Development Team. PX4 Autopilot. Available at: https://px4.io/, 2023. Accessed: 12 July 2023.

[5] Philipp Foehn, Elia Kaufmann, Angel Romero, Robert Penicka, Sihao Sun, Leonard Bauersfeld, Thomas Laengle, Giovanni Cioffi, Yunlong Song, Antonio Loquercio, et al. Agilicious: Open-source and open-hardware agile quadrotor for vision-based flight. *Science robotics*, 7(67):eabl6259, 2022.

[6] MathWorks. PX4 Support Package. Available at: https://it.mathworks.com/help/supportpkg/px4/, 2023. Accessed: 12 July 2023.

[7] PX4 Autopilot Development Team. PX4 Documentation: Simulation. Available at: https://docs.px4.io/main/en/simulation/, 2023. Accessed: 17 October 2023.

[8] PX4 Autopilot Development Team. PX4 Documentation: MAVROS Offboard Python Example. Available at: https://docs.px4.io/main/en/ros/mavros_offboard_python.html, 2023. Accessed: 17 October 2023.

[9] Wikipedia contributors. Travelling Salesman Problem. Available at: https://en.wikipedia.org/wiki/Travelling_salesman_problem, 2023. Accessed on 2023-11-06.