DIETI UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

TECHNICAL PROJECT FOR
FIELD AND SERVICE ROBOTICS EXAM

Motion planning and control for a VTOL UAV

Relatore:
Prof. Fabio Ruggiero

Candidato:
Michele Marolla
Matricola M58000251

Anno Accademico 2020–2021

Contents

Chapter 1

Introduction

The aim of this project is to develop a system able to perform motion planning and control for a **VTOL UAV**. In particular, an *AscTec Hummingbird* quadcopter has been considered; however, the system can work with any other UAV, as long as its dynamic parameters are known, a 3D model is available, and a control through total propeller thrust and torques is provided.

In the following, all the project specification are summarized.

Motion planning is performed via **artificial potential** method, where the force field is seen as a velocity vector. In order to avoid possible local minima, a **navigation function**-based algorithm has been implemented.

The control is achieved through a **passivity-based hierarchical controller**, with estimator of external wrench and unmodeled dynamics. The controller works at a fixed frequency of 1 kHz; moreover, it must be able to deal with uncertainty about the knowledge of UAV's dynamic parameters. In detail, the mass is underestimated by 10%, as well as the inertia parameters. In addition, a constant external force of 1 N is apply in each moment along one of the axis on the world frame.

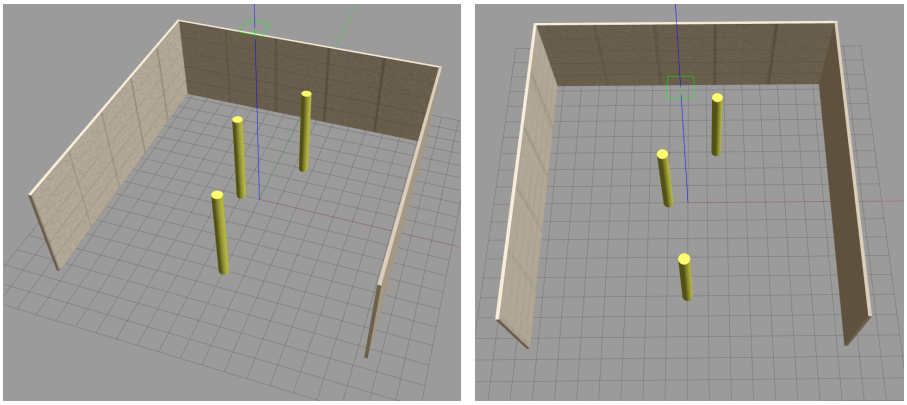
It is assumed to know the exact position and orientation of the robot.

The project has been developed in C++, using the **ROS** (Robot Operating System) framework, along with the **RotorS MAV** simulator, and the **Gazebo** physical engine. Also, **RViz** has been used for visualization and debugging purposes, and **Plotjuggler** for data processing and plotting. The project has been tested on Ubuntu 18.04 LTS, with ROS Melodic and Gazebo 9.0.

1.1 Scenario

The working space is a closed square environment, 5 meters high; three circular pillars are placed freely in the environment. Their height is 5 meters, while their radius is arbitrarily chosen equal to 0.25 m.

The environment has been created using the building tool of Gazebo; the



(a) *Perspective view* (b) *Top view*

Figure 1.1: 3D scenario built for this project.

used scenario is reported in Figure 1.1. In particular, with respect to the world ENU frame, the centers of the three pillars have been placed in (0, -5), (-1, 0) and (1.5, 3).

1.2 quad_control package

All the code developed has been placed inside the **quad_control** ROS package. Overall, four different nodes has been developed, along with one Gazebo plugin and some other support classes and functions.

Notice that among these nodes, no one has been implemented as ROS service or action. Even if conceptually, it would be possible (if not recommended) to implement – for example – the planner as a ROS action server, every node communicates through the standard publisher/subscriber paradigm. This choice has been made with a view to the future improvement of this project. Implementing the planner as an action will block the Manager execution, waiting for the planning to complete. This is acceptable in the project's current state, in which the Manager does nothing more than starting the various nodes. However, this could be a problem if in the future it carries out other tasks simultaneously. Using only the standard topic-based communication leaves the maximum flexibility for future changes in every direction.

Each object is completely configurable via ROS parameters: all the params that the user can define will be explained in the next chapters.

In the following, each node and object is described.

1.2.1 Planning

The motion planning problem is solved by the **APPlanner2D** node. The planning is carried out using the artificial planning method, assuming that

the environment has been mapped in the form of a 2D occupancy map.

Two other support classes has been developed.

MapAnalyzer is an object that scans the map and identifies chunks of contiguous pixels as an obstacle, in order to make the planner able to correctly compute the repulsive force associated with each obstacle.

NavigationFunc is instead an object that implements a navigation function-based planning, used by the planner to exit from local minima.

Further details on how motion planning is actually realized can be found on Chapter 2.

1.2.2 Control

UAV control is performed trough the **Controller** node. This node implements a hierarchical controller, computing total propellers' thrust and torques. These signals are then sent to a Gazebo plugin that converts them in the actual propellers' speed, used to command the robot.

The **PController** node is realized as a subclass of Controller: overriding the functions that computes some of the control variables, the hierarchical control is performed along with an external wrench and unmodeled dynamics estimator, thus achieving a passivity-based control.

LP2Filter is a templated object that implements a 2nd order low-pass filter, used both to compute the derivatives of roll pitch and yaw angles for the control, and to smooth the estimate of the external wrench.

Further details on how UAV control is actually realized can be found on Chapter 3.

1.2.3 Other nodes

Finally, two other nodes complete the package.

WindPlugin is a simple Gazebo world plugin that applies a constant force on the robot model. The robot model and the force intensity and direction are configurable through ROS parameters. The plugin can be configured using the four parameters in Table 1.1.

Manager is a node that, at the current project state, just sends appropriate requests for motion planning based on the user-specified trajectory points. This object can be very useful considering further development of the project; in fact, it can be used for more complex or user-oriented task, e.g., performing real-time data plot and visualization, or interactive motion planning or robot control.

For planning the trajectory, you can just pass the parameters: `xi`, `yi`, `zi`, `yawi`, where $i = 0, 1, 2, ...$ for any arbitrary number of points. If the yaw is not selected, the previous value will be used (or zero, if no other values were defined).

Table 1.1: Wind plugin configurable parameters.

Name	Type	Default value	Description
wind/enabled	bool	false	If true, the plugin is enabled.
wind/fx	double	0.0	Force applied on the x axis.
wind/fy	double	0.0	Force applied on the y axis.
wind/fz	double	0.0	Force applied on the z axis.
wind/model	string		UAV's model name.
wind/link	string		UAV's base link name.

1.3 World building

The scenario has been built (and can be easily modified) directly into Gazebo, using its building features. Walls can be added or resized, as well as any other object inside the environment.

Actually, there is only one limitation in building the scene. The map scanning proceeds from the top-left corner to the bottom-right one, and the first obstacle detected is identified as the outer wall. This is only needed if you want to have a different range of influence for walls and obstacles, otherwise you can place walls wherever you want.

You can use the given scenario layout with the three walls on north, east and west, using the `scene.launch` that already fullfill this requirement, or otherwise build a new scene from scratch.

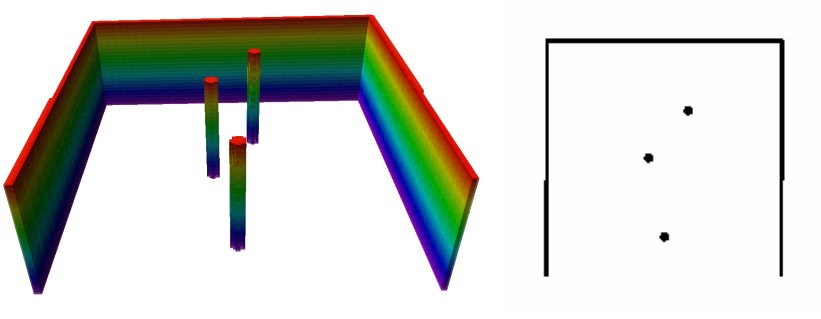
When the virtual scene is complete, a 3D octomap is built, using the **gazebo_octomap** plugin, provided by RotorS Mav simulator. The octomap is read by an **octomap_server** node, that also publish a 2D occupancy grid obtained as the projection of the octomap onto the ground. Finally, this map is saved to a file using a **map_saver** node, from map_server package. This procedure can be done by hand, or more easily through the procedure illustrated in the next section.

In Figure 1.2 is the 3D octomap and the resulting 2D occupancy grid, corresponding to the scenario in Figure 1.1.

Notice that at the current state, only a 2D planning is implemented. This means that any object in the scene is projected onto the ground, and the resulting map is used for planning. In other words, if a small object is placed in the scene, the UAV will not be able to fly over this object, even if in theory it could; instead, the planner will try to build a trajectory around it. This is better discussed in Chapter 4.

1.3.1 Generating octomap and occupancy grid from scene

If you want to integrate your own Gazebo scene in the project, it is needed to generate a 2D occupancy grid representing the scene itself. In order to speed



(a) *3D octomap* (b) *2D occupancy grid*

Figure 1.2: 3D and 2D retrieved by scanning the scenario.

up the procedure, two launch file and a Python script has been created.

To generate a 3D octomap from a Gazebo scene, you can use the `buildOcto` launch file, with the following syntax:

```
roslaunch quad_control buildOcto.launch world_name:=<your_world_name>
```

The resulting octomap will be saved into the `quad_control/worlds/octo` directory, and can be visualized using Octovis.

Then, you can generate a 2D occupancy grid from this octomap using the `buildMap` launch file, with the following syntax:

```
roslaunch quad_control buildMap.launch world_name:=<your_world_name>
```

The resulting map will be saved into the `quad_control/worlds/maps` directory. Notice that, in order to be properly integrated with the system, the .yaml map configuration file needs a few changes. This can be done automatically using the `fixMap.py` Python script, placed in `quad_control/script`, passing as argument the world name:

```
python fixMap.py <your_world_name>
```

Chapter 2

Motion planning

As it was stated in the previous chapter, motion planning is performed by the APPlanner2D node using the artificial potential method.

Planning can be requested sending on the `/planRequest` topic a PlanRequest message; it is a custom message type, defined as:

```
UAVPose[]               q
float64[]               steadyTime
nav_msgs/OccupancyGrid  map
```

The field `q` is a variable-length array of desired pose: the trajectory will be planned as a sequence of segment that connect q_i to q_{i+1}, whenever possible. The `UAVPose` is another custom message that contains UAV position and yaw angle (i.e., quadrotor flat outputs), defined as:

```
geometry_msgs/Point position
float64             yaw
```

For a UAV it is more useful to use such a custom message instead of standard `geometry_msgs/Pose` message, since only the yaw is taken into account for the planning problem.

The field `steadyTime` is a variable-length array that contains the amount of time that the UAV will spend in steady position, before executing the next segment. More specifically, the i-th value is the number of seconds the robot will remain hovering after executing the i-th segment, i.e., the segment connecting q_i and q_{i+1}. If no time is specified, i.e., the size of `steadyTime` is less than the size of `q`, the corresponding steady time is assumed to be zero.

Finally, the field `map` contains the 2d occupancy grid of the environment. This is typically retrieved using a **map_server** node that reads the map from a local file. The map is the scanned using by a MapAnalyzer object. However, if the map has already been scanned, and a new instance of the planning problem needs to be solved in the same environment, it is possible

to send an empty map (i.e., either its width or height must be set to zero). In this case, the scanning phase is ignored, saving execution time.

The planned trajectory is published as a Trajectory custom message, defined as an array of TrajectoryPoint:

```
TrajectoryPoint[] points
```

A TrajectoryPoint is defined as:

```
UAVPose              p
geometry_msgs/Accel  v
geometry_msgs/Accel  a
float64              t
```

where `p` is the UAV pose, `v` and `a` are UAV's velocity and acceleration, and `t` is the used sample time.

It is possible to publish the trajectory as a standard `nav_msgs/Path` message too, in order to visualize it in RViz.

2.1 Planning via artificial potentials

The planning is realized via the artificial potential methods, where forces are seen as velocity vectors applied on the robot. This can be used for on-line planning, gathering information about the environment from UAV sensors. For this application, it is assumed that the environment has already been mapped, and a 2D occupancy grid is at disposition.

Let q_g be the goal configuration, q the robot configuration, and $e(q) = q_g - q$ the error. The **attractive potential** is defined, using a paraboloid expression when $\|e\| < 1$ and a conical expression when $\|e\| > 1$; the corresponding attractive force is:

$$f_a(q) = \begin{cases} k_a e(q), & \|e(q)\| \leq 1 \\ k_a \dfrac{e(q)}{\|e(q)\|}, & \|e(q)\| > 1 \end{cases} \tag{2.1}$$

where $k_a > 0$.

Now, let $\mathcal{CO}_i$ be the convex i-th $\mathcal{C}$-obstacle, $\eta_{0,i}$ the range of influence of the i-th obstacle, $\eta_i(q) = \min_{q' \in \mathcal{CO}_i} \|q - q'\|$ the distance of q to $\mathcal{CO}_i$, and $\gamma = 2, 3$. The repulsive force performed by the i-th obstacle on the robot is defined as:

$$f_{r,i}(q) = \begin{cases} \dfrac{k_{r,i}}{\eta_i^2(q)} \left(\dfrac{1}{\eta_i(q)} - \dfrac{1}{\eta_{0,i}}\right)^{\gamma-1} \nabla \eta_i(q), & \eta_i(q) \leq \eta_{0,i} \\ 0, & \eta_i(q) > \eta_{0,i} \end{cases} \tag{2.2}$$

where $k_{r,i} > 0$.

Thus, the total force applied on the robot is:

$$f_t(q) = f_a(q) + \sum_{i=1}^{p} f_{r,i}(q) \tag{2.3}$$

and it is seen as a velocity vector moving the robot, so $\dot{q} = f_t(q)$.

To retrieve the robot configuration, the Euler integration method is applied, leading to $q_{k+1} = q_k + T f_t(q_k)$, where T is the integration step.

To improve the quality of the path, a variable T has been used. In particular, the user can define two quantities: `sampleMin` and `sampleMax`. If the nearest obstacle has a distance from the robot of at least two times the maximum range of influence, or the goal is very distant from the robot, `sampleMax` is used. If instead the distance from the obstacle is less than before, but greater the range of influence, or the goal is not so distant, the average between `sampleMin` and `sampleMax` is used. Finally, if the robot is inside the obstacle's range of influence or near the goal, `sampleMin` is used. Obviously, the range of influences and the goal distance thresholds can be defined by the user.

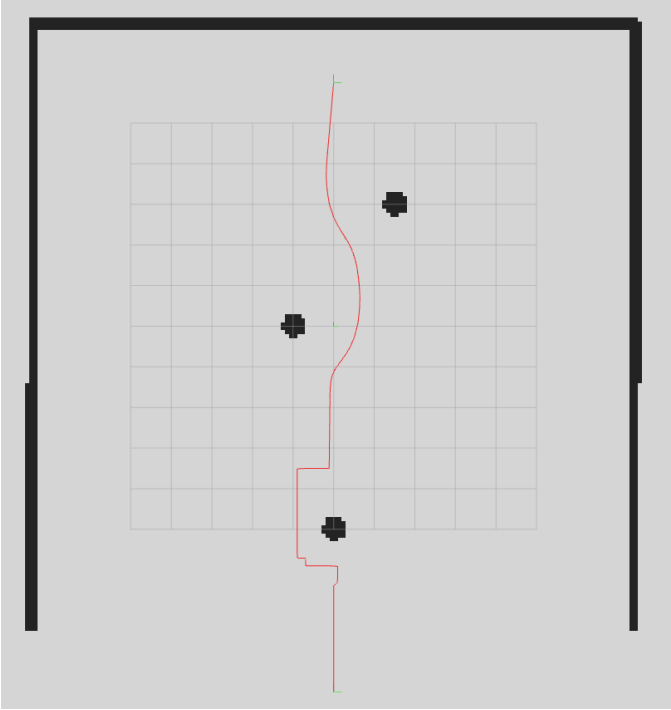


Figure 2.1: Planned trajectory.

In Figure 2.1, the trajectory generated by the planner is represented. The environment is the one described in the previous chapter. The trajectory was request to connect the points $A(-9, 0, 0)$, $B(-9, 0, -1)$, $C(6, 0, -1)$ and

$D(6, 0, 0)$, represented by the frames in the image. Notice the different yaw angles, equal to 0 in the spawn point A, $-\pi/2$ in B, and 0 again in C and D. As you can see, the trajectory is bended around the obstacles, thus avoiding collisions.

In order to overcome the first pillar, it is necessary to exit from a local minima: details are presented in the next section.

Notice that the planning request is expressed with respect to the world NED frame.

2.2 Local minima problem: navigation functions

In order to tackle the local minima problem, i.e., $f_t = 0$ when $\|e(q)\| \neq 0$, a navigation function-based algorithm has been implemented.

This choice has been preferred to other approaches, like best-first algorithm, for its efficiency. In fact, since it is assumed that the environment is known, navigation functions are always able to correctly plan a trajectory that lead the robot outside the local minimum. Best-first algorithms, on the other hand, do not assure to avoid the same (or other) local minima.

The algorithm is implemented through the NavigationFunc class.

For efficiency reasons, when the planner detects that the trajectory is stuck in a local minima, an environment submap is formed. This is a map of only a portion of the whole world, shaped as a square with adjustable length, and the current robot position as the center.

The resolution of this submap can be chosen by the user as a multiple of the occupancy grid resolution.

On this submap, the goal cell is identified and a navigation function is computed. Finally, the path connecting the starting cell with the goal cell is appended to the actual trajectory.

When the goal or the submap boundary is reached, i.e., the robot exited the local minimum, the planning continues as usual.

Since the path resulting from the navigation function can be made of several segments with different directions, the user can specified sample time and maximum velocity specifically used in this scenario.

In Figure 2.2, it can be seen the use of navigation function to overcome the first pillar.

Starting from the bottom of the image, in proximity to the point highlighted in purple, a local minimum is found: the trajectory moves in the centroid of the nearest cell, and an appropriate navigation function is built.

The goal inside the navigation function is identified in the point highlighted in blue, and a path from the current UAV's position is found.

Finally, when the goal point is reached, the planning continues through the usual artificial potential method.

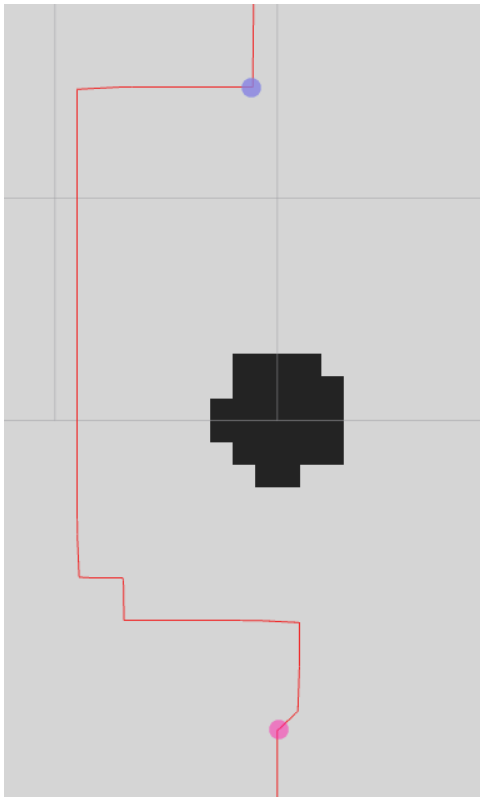


Figure 2.2: Navigation function in the planned trajectory.

2.3 Configurable parameters

The Table 2.1 contains all the configurable parameters for the planning nodes, in alphabetical order. If one parameter is not explicitly specified in the ROS parameter server, its default value will be used.

Table 2.1: Planner configurable parameters.

Name	Type	Default value	Description
showPath	bool	false	Visualize trajectory in RViz.
showPathPoints	bool	false	Visualize trajectory points in RViz.
eta	double	1.0	Obstacles range of influence. If this is specified, it is etaObst = etaWall = eta.
etaObst	double	1.0	Obstacles range of influence.
etaWall	double	1.0	Walls range of influence.
gamma	double	2.0	$\gamma = \{2, 3\}$, used for repulsive forces.
goalDistAvg	double	0.1	If distance from goal is greater than goalDistAvg, sampleMax sample time is used.
goalDistMin	double	0.05	If distance from goal is less than goalDistAvg and greater than goalDistMin, sampleMin sample time is used.
ka	double	1.0	Attractive forces gain.
kr	double	1.0	Repulsive forces gain. It is assumed $k_{r,i} = k_r, \forall i$.
maxVel_o	double	0	Maximum angular velocity (yaw). If 0, it is unlimited.
maxVel_p	double	0	Maximum linear velocity. If 0, it is unlimited.
maxVerticalAcc	double	5.0	Maximum vertical acceleration.
navErrTolerance	double	0.005	Error tolerance while planning with navigation functions.
navEta	double	0.5	Obstacles range of influence while planning with navigation function.
navFuncRadius	double	3.0	Submap size used for navigation functions.
navMaxDisp	double	0.01	Maximum q displacement that identifies a local minimum.
navMaxFt	double	0.1	Maximum velocity norm that identifies a local minimum.
navRatio	int	1	Ratio between navigation function resolution and occupancy grid resolution.
navSampleTime	double	sampleTimeMin	Sample time used with navigation functions.
navVelocity	double	1.0	Maximum velocity while planning with navigation functions.
o_eps	double	0.01	Maximum acceptable orientation error [rad].
p_eps	double	0.01	Maximum acceptable position err [m].
qDiffMin	double	0.0	Maximum q displacement in one simulation step while using sampleTimeMin.
qDiffMax	double	0.0	Maximum q displacement in one simulation step while using sampleTimeMax.
rate	double	100.0	Node execution frequency.
sampleTime	double	0.01	Trajectory sample time. If this is specified, constant sample time is used.
sampleTimeMin	double	0.001	Trajectory minimum sample time.
sampleTimeMax	double	0.01	Trajectory maximum sample time.

Chapter 3

UAV Control

UAV motion control is performed using a passivity-based control approach, with an external wrench and unmodeled dynamics estimator.

Actually, two different nodes have been implemented: **Controller** is a simple hierarchical control, while **PController** extends it introducing the estimator. This choice lead to a very clear separation of the hierarchical control logic and the estimator, and completely removes any repeated code.

As introduced in the first chapter, control is actuated publishing the propellers' total thrust u_T and torques τ^b. These signals are converted into the actual propellers' speed by the **ned_plugin**.

In the following sections, the control implementation is discussed in detail.

3.1 Hierarchical control

The hierarchical controller can be represented as in Figure 3.1. It consists of two feedback loops:

- The outer loop takes as input the desired position $p_{b,d}$, velocity $\dot{p}_{b,d}$ and acceleration $\ddot{p}_{b,d}$ given by the planner and the actual position p_b and velocity $\dot{p}_b$ provided by odometry sensors; the outputs are the thrust u_T and the desired roll and pitch angles.
- The inner loop takes as input the desired attitude $\eta_{b,d}$, where the yaw is provided by the planner, roll and pitch by the outer loop, and the actual attitude η_b provided by odometry sensors; torques τ^b are computed as output.

In particular, in the outer loop, the virtual acceleration input μ_d is computed as:

$$\mu_d = \begin{bmatrix} \mu_x \\ \mu_y \\ \mu_z \end{bmatrix} = -K_p \begin{bmatrix} e_p \\ \dot{e}_p \end{bmatrix} + \ddot{p}_{b,d} \tag{3.1}$$

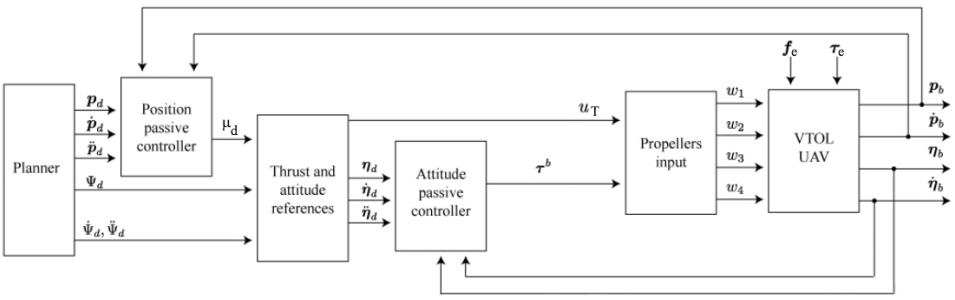


Figure 3.1: Hierarchical control block scheme.

where $e_p = p_b - p_{b,d}$ and $\dot{e}_p = \dot{p}_b - \dot{p}_{b,d}$, and an integral action can been added to; from this we get:

$$u_T = m\sqrt{\mu_x^2 + \mu_y^2 + (\mu_z - g)^2} \tag{3.2}$$

$$\phi_d = \sin^{-1}\Big(m\frac{\mu_y \cos\psi_d - \mu_x \sin\psi_d}{u_T}\Big) \tag{3.3}$$

$$\theta_d = \text{atan2}(\mu_x \cos\psi_d + \mu_y \sin\psi_d, \mu_z - g) \tag{3.4}$$

In the inner loop, the virtual torque is designed as:

$$\tilde{\tau} = -K_o \begin{bmatrix} e_\eta \\ \dot{e}_\eta \end{bmatrix} + \ddot{\eta}_{b,d} \tag{3.5}$$

where $e_\eta = \eta_b - \eta_{b,d}$, $\dot{e}_\eta = \dot{\eta}_b - \dot{\eta}_{b,d}$, and again an integral action can be considered; from this:

$$\tau^b = I_b Q \tilde{\tau} + Q^{-T} C(\eta_b, \dot{\eta}_b)\dot{\eta}_b \tag{3.6}$$

where I_b is the inertia matrix, $C(\eta_b, \dot{\eta}_b)$ is the Coriolis matrix, and Q is defined as:

$$Q = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix}$$

3.2 Passivity-based control with wrench estimator

With these control scheme, the feedback linearization in the angular part is avoided, and an estimator is used to compensate for the unknown wrench (i.e., external disturbances, unmodeled dynamics, parametric uncertainties, and so on).

Scheme block is reported in Figure 3.2.

3.2.1 External wrench estimation

Let q b the generalized momentum vector. Consider the RPY quadrotor dynamic model in compact form:

$$M_\xi \ddot{\xi} + C_\xi \dot{\xi} + G_\xi = \Delta u + F_e \tag{3.7}$$

Assuming $\hat{F}_e(0) = 0$, the external wrench is estimated as:

$$\hat{F}_e(t) = \begin{bmatrix} \hat{f}_e \\ \hat{\tau}_e \end{bmatrix} = k_0 q - \int_0^t \Big[c_0 \hat{F}_e(\sigma) + k_0 (C_\xi^T \dot{\xi} + \Delta u - G_\xi) \Big] d\sigma \tag{3.8}$$

3.2.2 Motion control

The only difference between this and the previous control scheme is the computation of the virtual acceleration and the torques vector. Without deepening in formal and mathematical details, for this controller scheme the virtual acceleration is:

$$\bar{\mu} = \mu_d - \frac{1}{m} \hat{f}_e, \quad \mu_d = \ddot{p}_{b,d} - \frac{1}{m} (K_P e_p + K_D \dot{e}_p) \tag{3.9}$$

while the torques are computed as:

$$\tau^b = Q^{-T} (M(\eta_b) \ddot{\eta}_r + C \dot{\eta}_r - \hat{\tau}_e - D_O \nu_\eta - K_O e_\eta) \tag{3.10}$$

where $\nu > 0$ is a coupling parameter, K_P, K_D, K_O, D_O are positive definite matrices, and the following quantities are defined:

$$\dot{\eta}_r = \dot{\eta}_{b,d} - \nu e_\eta, \quad e_\eta = \eta_b - \eta_{b,d} \quad \ddot{\eta}_r = \ddot{\eta}_{b,d} - \nu \dot{e}_\eta, \quad \dot{e}_\eta = \dot{\eta}_b - \dot{\eta}_{b,d}, \quad \nu_\eta = \dot{e}_\eta + \nu e_\eta$$

3.3 Implementation details

The hierarchical control is implemented in the Controller node.

The controller is initialized and waits for odometry data and a trajectory to be planned. When a trajectory is ready, at each simulation step the controller retrieved the current desired trajectory point, and compute propellers' thrust and torques accordingly.

One thing to be noticed is that the virtual acceleration input μ_d and the torque vector τ are computed using two separate functions, defined as *virtual functions*.

In this way, the passivity-based controller (the PController node) can be implemented as a Controller derived class, that only modifies these two functions.

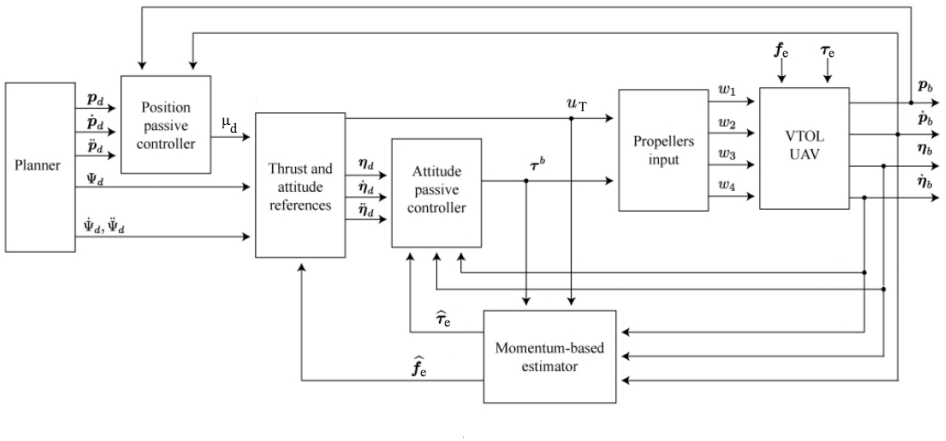


Figure 3.2: Passitivy-based control block scheme.

Also, the desired attitude compute by the outer loop is derived and filtered using the LP2Filter class discussed in Chapter 1. The same is used to filter the estimated wrench in the case of passivity-based control.

Regarding the signal filtering, in order to avoid the transient behaviour of the filter, the signal is processed for a certain number of steps, configurable by the user.

3.4 Configurable parameters

The Table 3.1 contains all the configurable parameters for the planning nodes, in alphabetical order. If one parameter is not explicitly specified in the ROS parameter server, its default value will be used.

3.5 Simulations and results

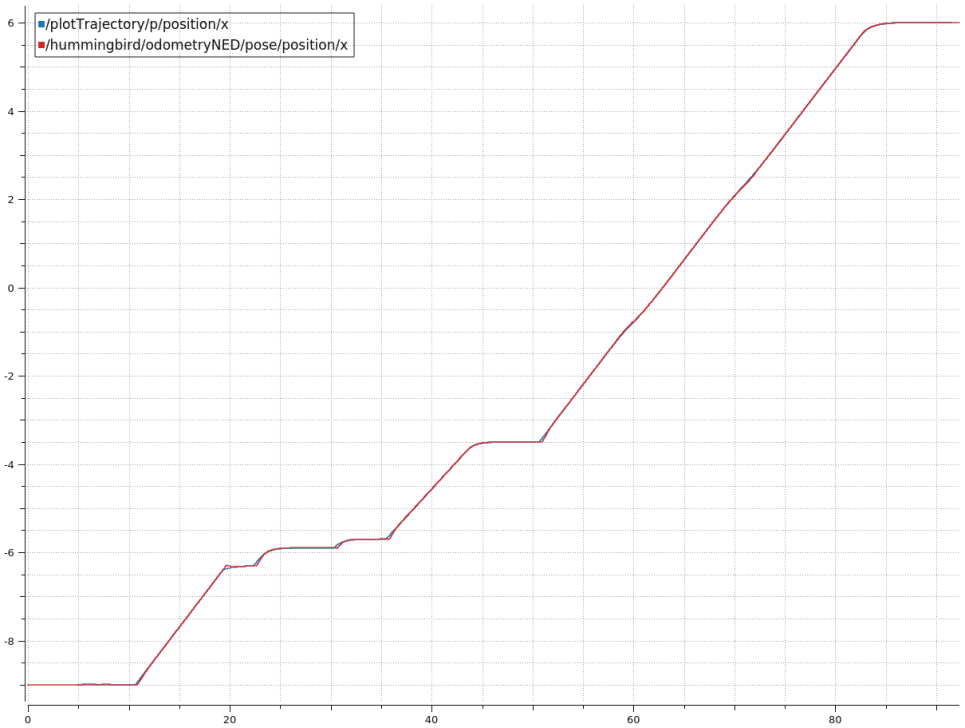
The main simulation can be executed running the `main.launch` file. The scenario is the one represented in Chapter 1, with the trajectory planned in Chapter 2. A video showing the simulation can be found in the `quad_control/doc` directory.

In Figure 3.3 and 3.4 the simulation's results are presented. Notice that, within a certain error threshold, the UAV follows the planned trajectory in terms of x, y and z coordinate and the yaw angle. In Figure 3.5 are reported the position and orientation error.

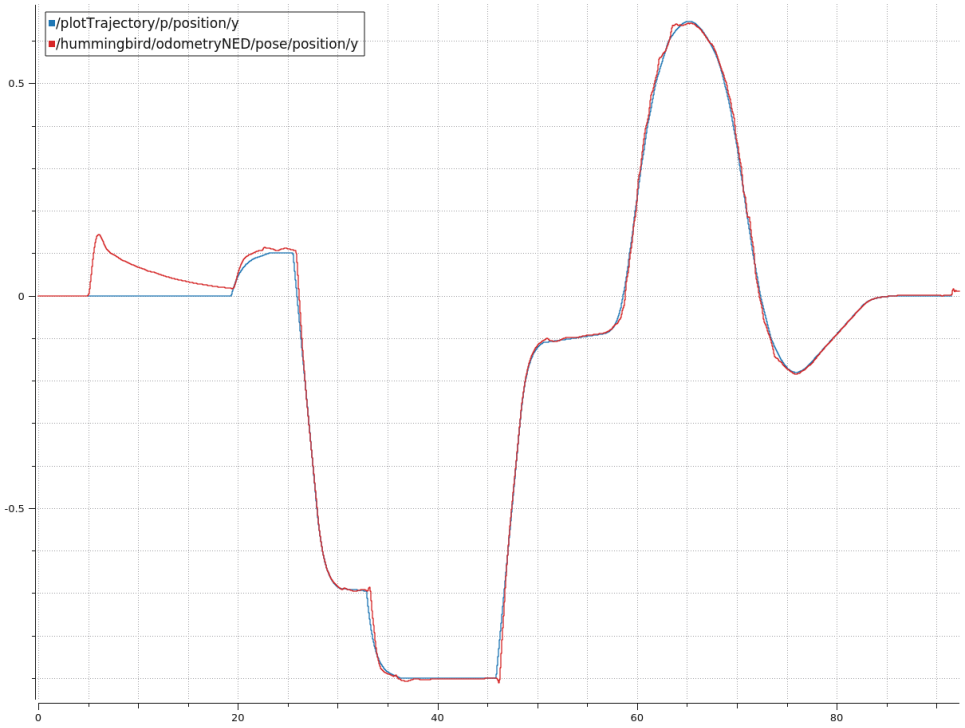
Excluding the take-off phase, the maximum positional tracking error norm is less than 9.5 cm circa; the maximum error on the yaw angle is instead approximately 0.08 rad $\approx$ 4.6 deg.

Table 3.1: Controller configurable parameters.

Name	**Type**	**Default value**	**Description**
c0	double	1000.0	Estimator parameter.
estFilterRate	double	10000.0	Rate of the estimator low-pass filter.
estimator	bool	true	Enable/disable the wrench estimator.
estFilterBand	double	10.0	Band for the estimator filter.
filterRate	double	50000.0	Rate for the attitude derivative filter.
Ibx	double	0.007	UAV's inertia along x axis; default is *Hummingbird* nominal value.
Iby	double	0.007	As before, on y axis.
Ibz	double	0.012	As before, on z axis.
ko	double	100.0	Angular gain.
koi	double	0.0	Angular integral gain.
kod	double	1.0	Angular velocity gain.
kp	double	100.0	Position gain.
kpi	double	0.0	Position integral gain.
kpd	double	10.0	Linear velocity gain.
k1	double	100.0	Bandwidth of the first derivative filter.
k2	double	100.0	Bandwidth of the second derivative filter.
landing	bool	false	If true, the UAV will land after completing the trajectory; if false, it will stay in hovering.
m	double	0.68	UAV's mass; default is *Hummingbird* nominal value.
plotTrajectory	bool	false	Published the desired position (used for plotting).
rate	double	1000.0	Node execution frequency.
v	double	100.0	Param ν for passivity-based controller.

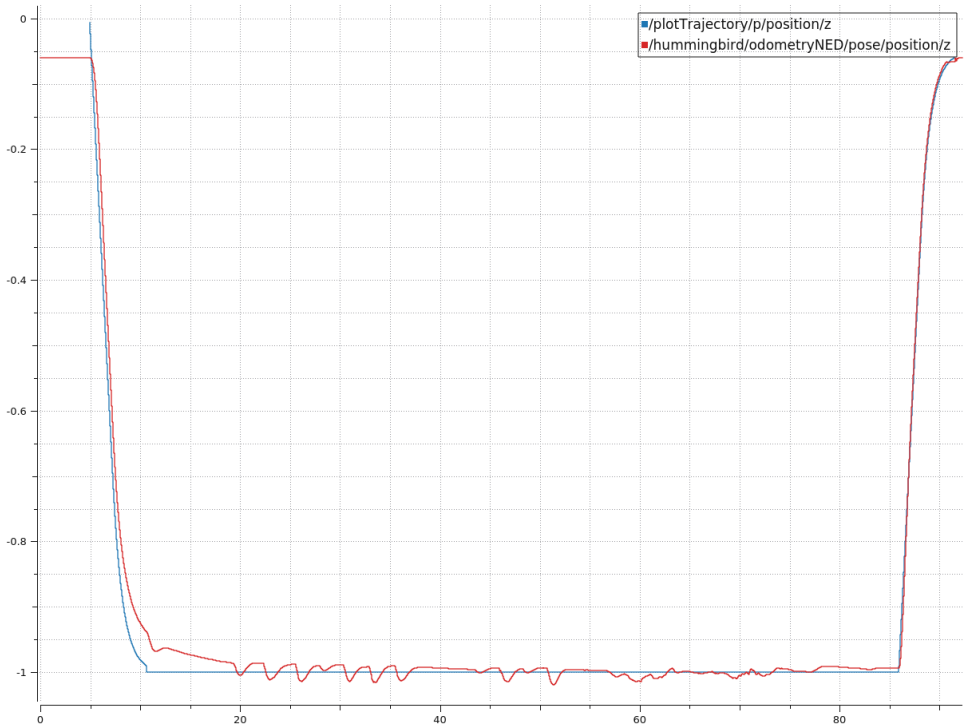


(a) *X coordinate*

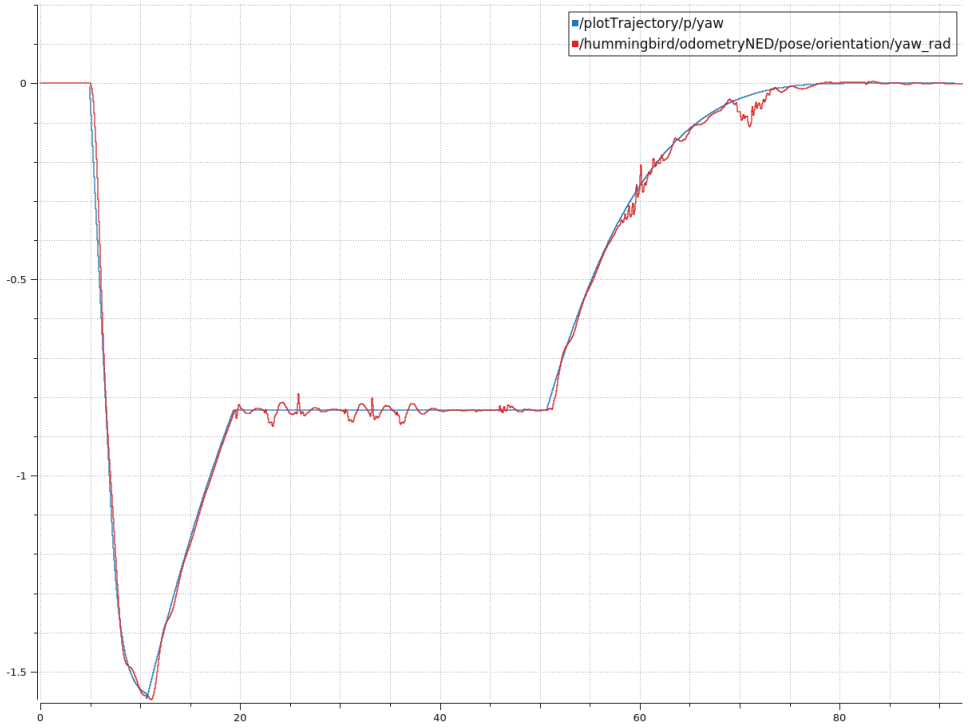


(b) *Y coordinate*

Figure 3.3: Planned and effective x and y position.

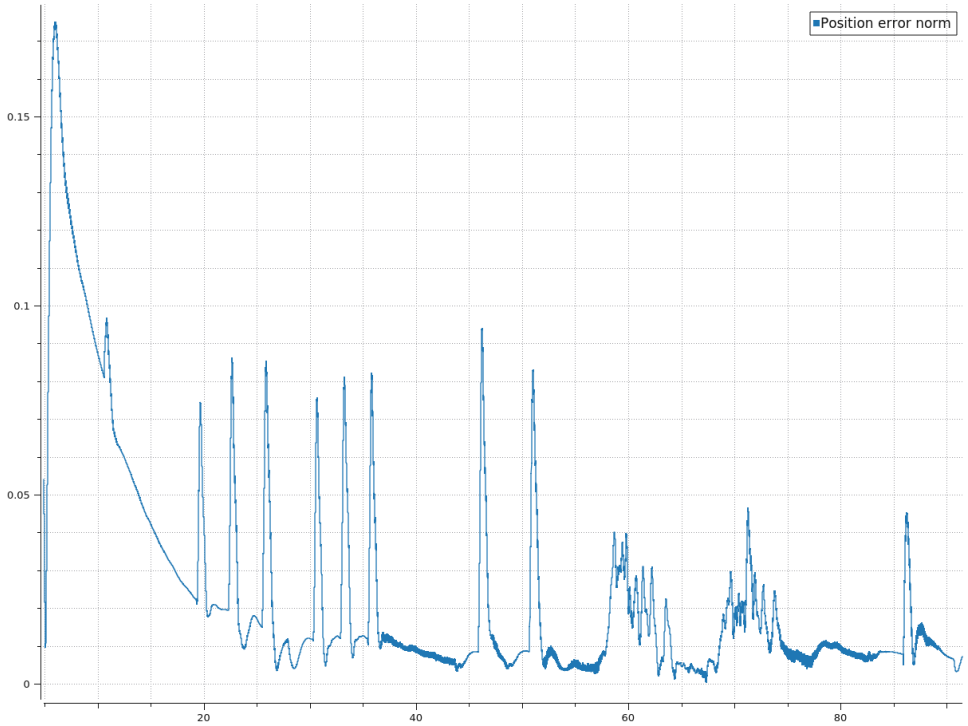


(a) *Z coordinate*

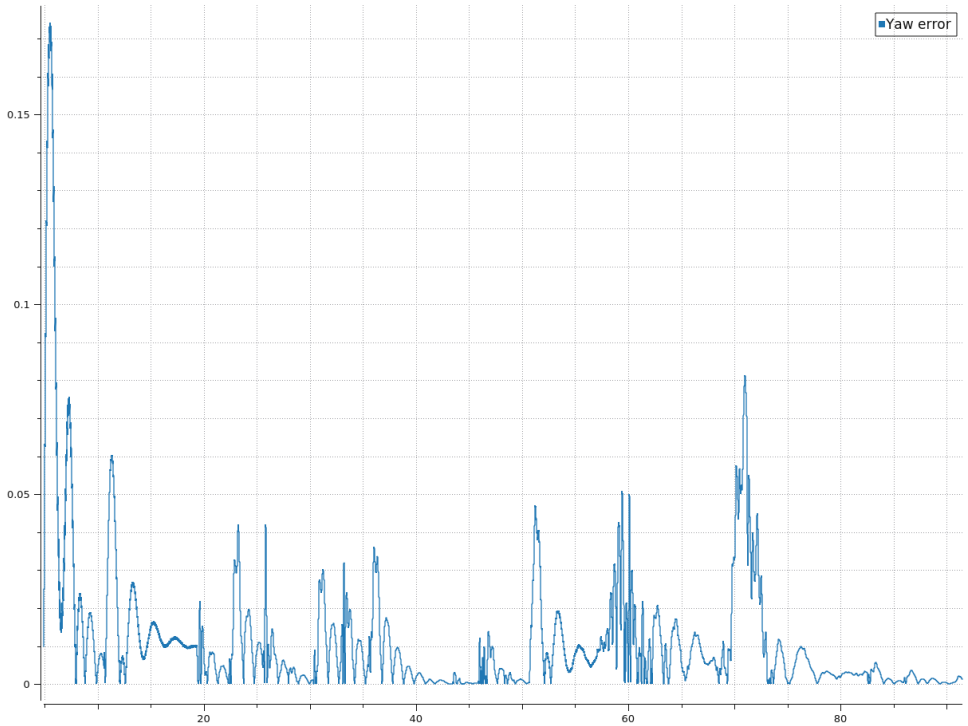


(b) *Yaw angle*

Figure 3.4: Planned and effective z position and orientation.



(a) *Position error norm*



(b) *Yaw error*

Figure 3.5: Position and orientation error.

Chapter 4

Conclusions

The main objective for this project was to develop a system able to perform motion planning and control for a VTOL UAV, taking into account parameter uncertainties and external disturbances. These requirements have been fulfilled, achieving a good results in terms of trajectory planning and tracking error.

Even if the results seems pretty good, there are lot of possible improvements and additional features that can be added. These things have not been implemented yet since they were outside the scope of this project, due to complexity or required time.

Possible improvements include:

- UAV landing is now achieved planning a trajectory from the current altitude to a point close to the ground ($z = 0.05m$), and then the propellers are turned off. This works but it's not the best solution: the UAV shows in fact a little oscillation, and could be improved to obtain a smoother behaviour.
- Navigation function algorithm currently uses a fixed sample time equal to `navSampleTime`: a variable sample time could improve the trajectory.
- 3D planning could be implemented, using the octomap instead of the occupancy grid. The basic code structure will be the same, but the obstacles have to be considered in the three spatial dimensions.
- For unknown environments, world mapping could be implemented simultaneously to trajectory planning using on-board sensors, i.e., LIDARs or depth cameras.