Automation and Robotics Engineering

FIELD AND SERVICE ROBOTICS (FSR)

Motion planning and control of a quadrotor UAV

Instructor:
Fabio Ruggiero

Students:
Antonio Rescigno
P38000047
Francesco Grasso
P38000046

Accademic Year 2021/2022

Index

Abstract

The report is about the Quadrotor analysis in terms of motion planification, dynamics modelling and control.

Different approaches with comparable goals and results will be explained. At first, the planner of a trajectory is generated and regulated by the A* algorithm. About the dynamics analysis two methods are used to describe it. A RPY model and a Mass-Damper-Spring like model for the linearized system around an equilibrium point are built. The first will be involved into the passivity-based control scheme without observation of external disturbances. The second, on the other hand, is applied for PID controllers to regulate attitude and position. In this case a Kalman filter is useful in terms of disturbances estimation. Moreover, the estimation measured are applied as input in an admittance controller. The admittance control output will improve the tracking goal by evaluating the interaction forces. Plots and figures will be the correlation which explains the results.

Chapter 1

Motion Planning

A classical problem for robotic navigation is how to efficiently navigate from one point to another and what to do if obastacles are encountered along the way. Many map based path planning algorithms attempt to solve this problem, all with varying levels of optimality and complexity. The A^* **algorithm** and the Probabilistic Roadmap method (**PRM**) are used in this chapter with the aim of finding a path for the UAV in a known environment with the presence of obstacles.

The Probabilistic Roadmap Method

The Probabilistic Roadmap Method gives as result an approximation of the configuration space. The basic idea is to take random samples from the configuration space of the robot, testing them for whether they are in the free space, and use a local planner to attempt to connect these configurations to other nearby configurations. Each rectilinear path between the two points is sampled and each sample is checked for collisions.

As shown in figure,the **occ.map.m** function creates a 3D map of a fixed size and with obstacles inside. It also deals with the plotting of the start and goal positions.

The line **setOccupancy(omap3D,xyzObstacles,1)** allows to define the positions occupied by obstacles by associating the number 1 if the position is occupied, -1 otherwise.

The same operation is replicated for the floor, considered as an obstacle.

The function **q.rand.gen.m** generates a fixed number of random configurations (established by the parameter **iter**). These points, if they do not collide with any obstacle, are added to the vector representing a subset of C_{free}. Precisely for this reason, as mentioned above, the **PRM** identifies an approximation of C_{free}. Also note, how the first and last points of the **points.rdmap** vector are associated to the start and goal point, respectively.

In the function **seg.gen.m**, for each point of the **points.rdmap** vector and therefore for each configuration of the approximation of C_{free} the neighbors (**qnear**) are calculated through the parameter **ds**: the distance of each point from all the others is calculated. Therefore if the distance is smaller of the threshold **ds**, the neighbors of the i-th point are added in the second column of the vector **indx**.

The algorithm ensures that the configurations subsequent to the first do not consider as neighbors the points from which they have already been considered as neighbors. In order to avoid useless calculations for the segments and for the plots that make the algorithm less performing: associations such as: 1- 10 and 10-1 are avoided, in fact if the path from the first to the tenth point is already present, the backward path, from the tenth point to the first is not necessary.

The Probabilistic Roadmap Method requires that the segment joining each pair of neighbors is appropriately sampled and that no point on it collides with an obstacle: this part of the code is developed by the function **segm.m**, which for each pair of neighbors and therefore for each pair of points whose indices are on the lines of the vector **indx** samples the segment joining the two points.

The segments that intersect an obstacle are then discarded while the extremes of the rest are added to the **tree** vector: the **tree** vector therefore contains all the points and their neighbors that the drone can reach in a straight line without colliding.

The A^* Algorithm

Also others algorithms, such as the Breadth-first search (**BFS**) or the Depth-first search (**DFS**), can be associated with the **PRM**.

In this case we have chosen to use the A^* algorithm which, through weighted arcs and a heuristic function, aims to search for the path at the lowest cost from a starting point to an end point.

During the implementation, the heuristic function was set to 0, with this assumption the A^* algorithm is better known as **Dijkstra's algorithm**.

The adjacency matrix was then constructed taking into account the connections obtained with the rows of the **tree** vector and the generic value of an element of the matrix was set equal to the distance between the i-th and j-th points. Of course, if the points are not connected, the value is set equal to 0.

The algorithm then ends with the calculation of the **tree.dfs** vector, which contains the position of the nodes to reach.

Of course, the algorithm does not necessarily end successfully, the path may not be found and in this case, if subsequent iterations do not lead to any improvement, it is necessary to increase the **iter** or **ds** parameters. This is equivalent to increasing the number of iterations and therefore the possible configurations but also to increasing the number of neighbors, with the aim of obtaining more paths to reach the goal.

Simulations

As shown in the following figures, we have chosen to place six obstacles of different heights on a 200x200x200 map. In yellow it is possible to see the graph of the starting point and a generic end point, in green there are the segments obtained with the roadmap and in red the path obtained with the A^* Algorithm.

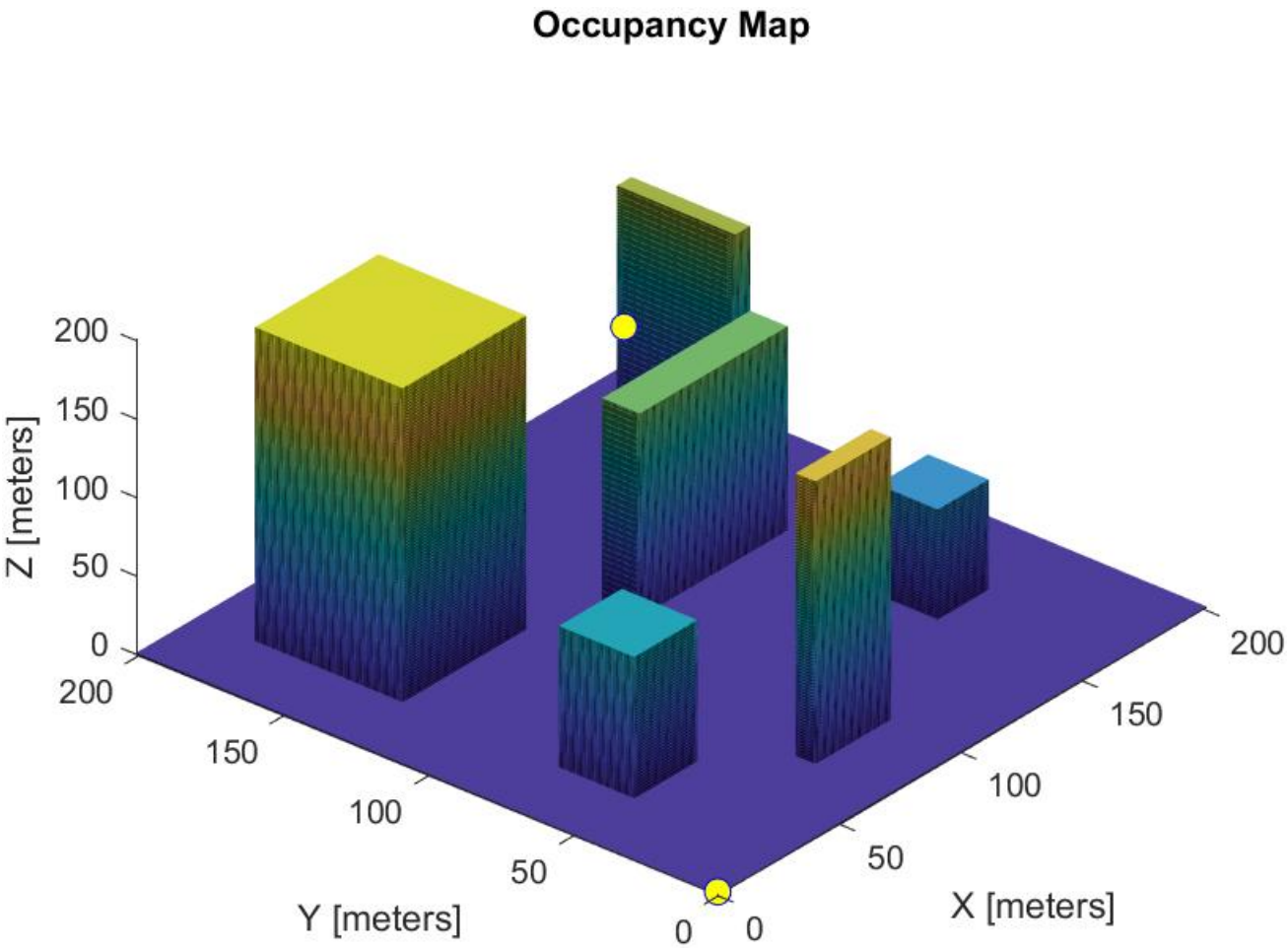


Figure 1.1: Occupancy Map

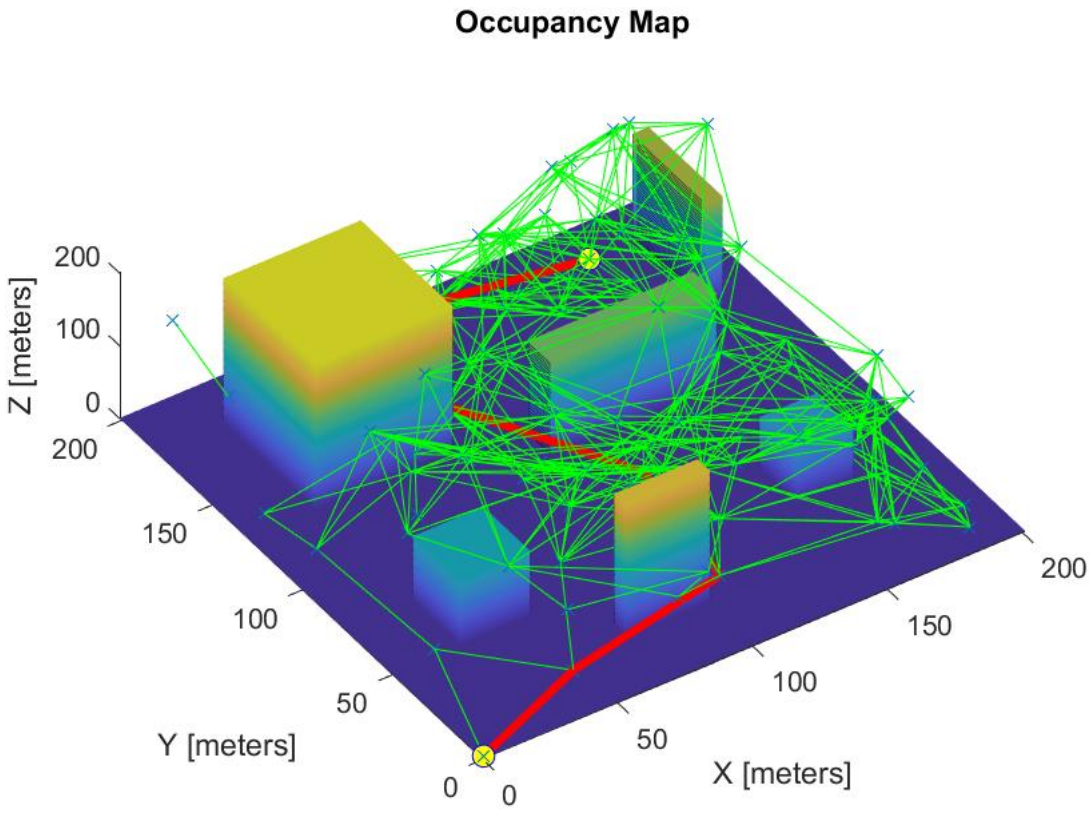


Figure 1.2: PRM and A^* with iter = 110 and ds = 70

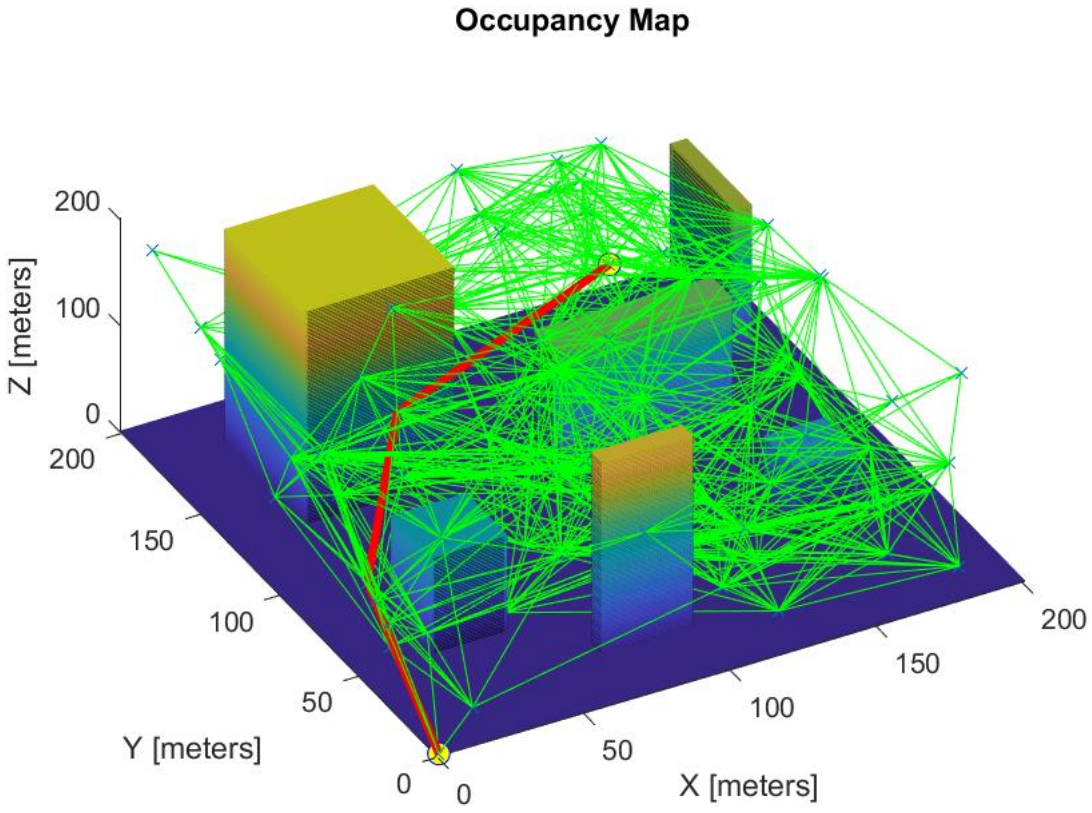


Figure 1.3: PRM and A^* with iter = 100 and ds = 100

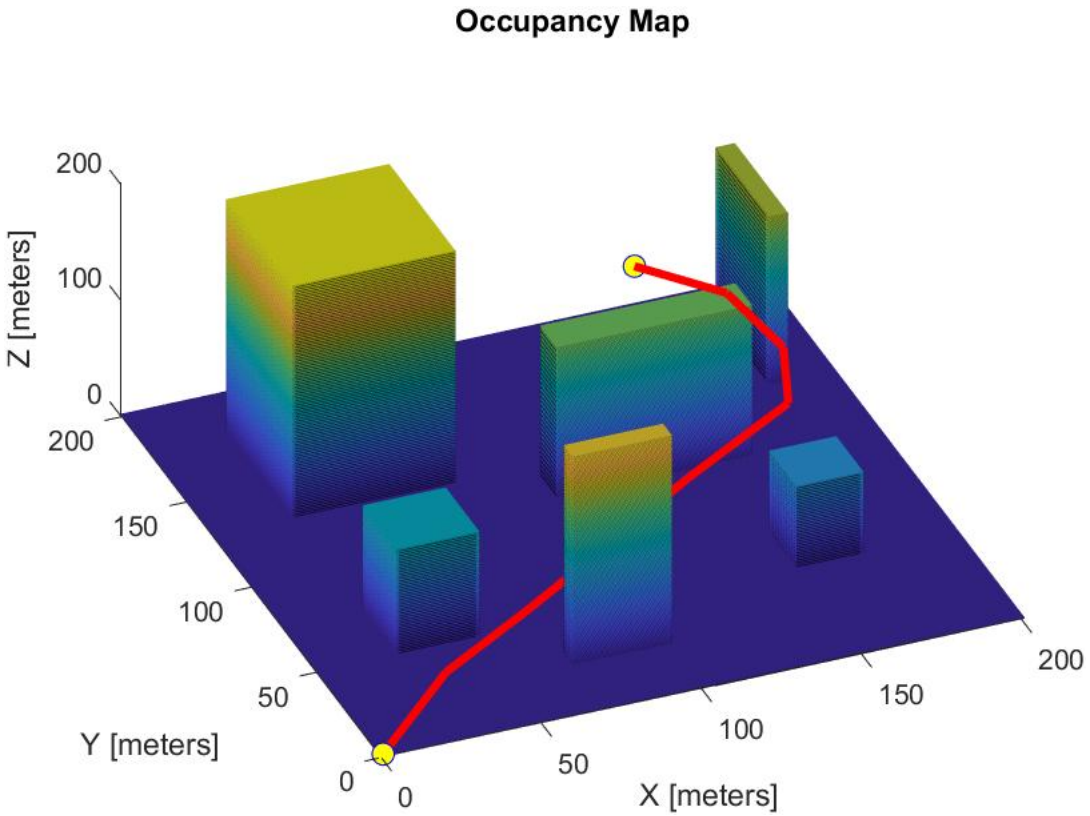


Figure 1.4: Path obtained with iter = 350 and ds = 50

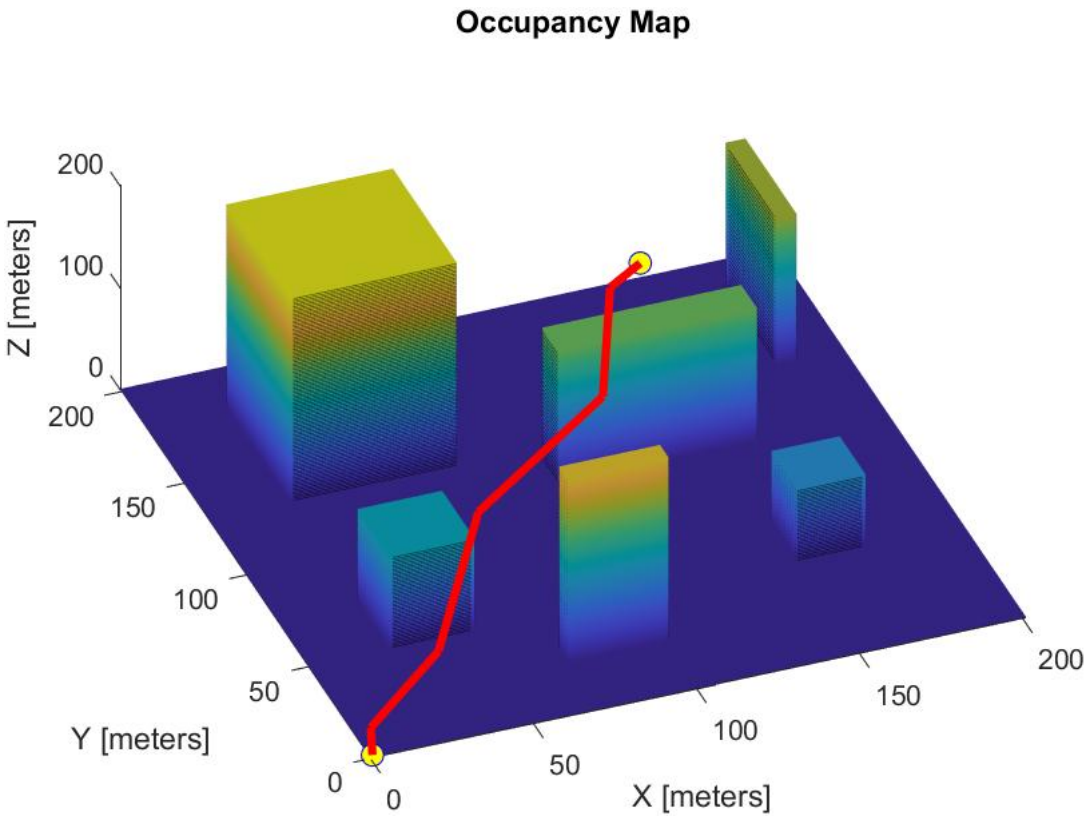


Figure 1.5: Path obtained with iter = 350 and ds = 70

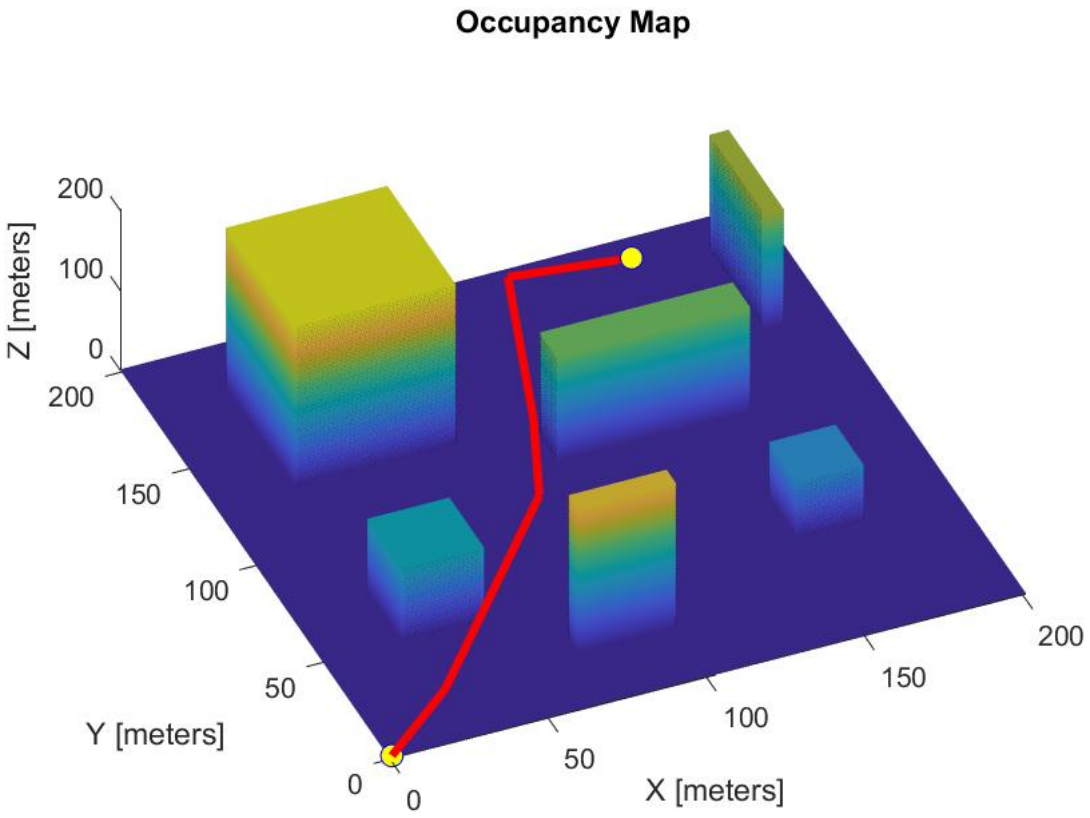


Figure 1.6: Path obtained with iter = 400 and ds = 100

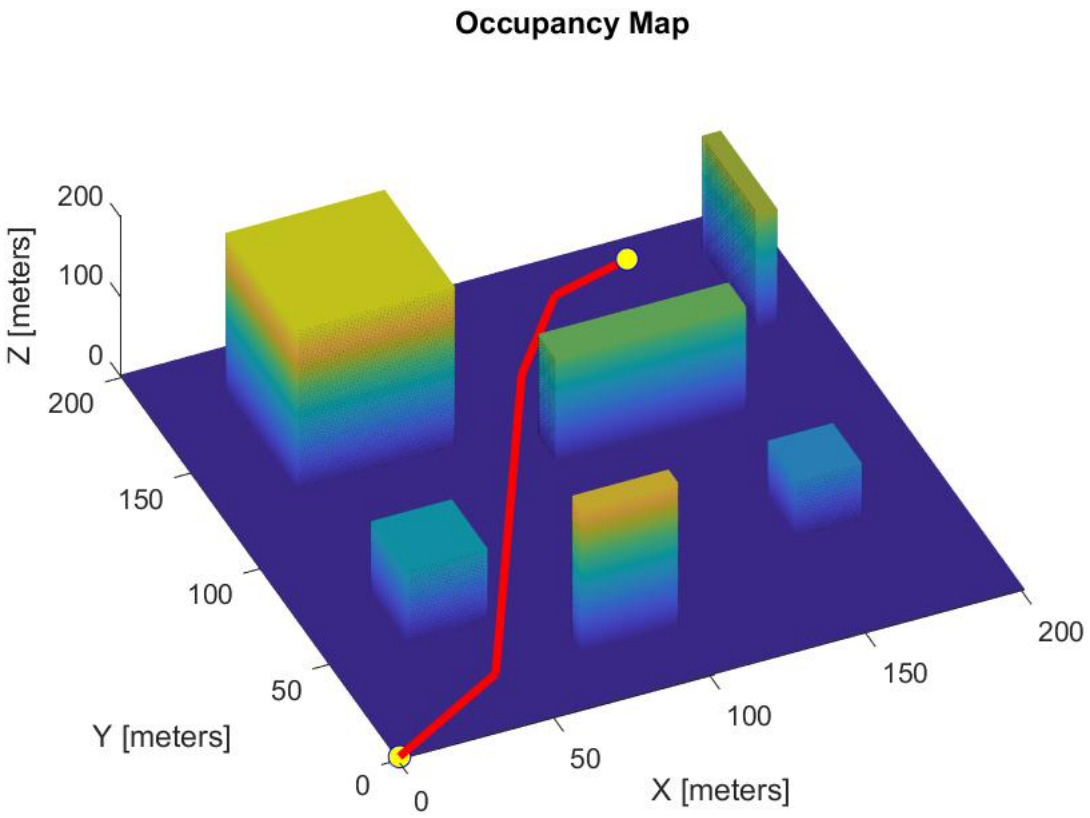


Figure 1.7: Path obtained with iter = 500 and ds = 150

As seen in the initial simulations, the path is much longer than in the last ones, this is due to very low values of **iter** and of **ds** ; it should be remembered that the algorithm A^* does not necessarily give a result: it often does the opposite (indicated by the algorithm with the string "Solution not found!") when the values of **ds** or of **iter** are around 50- 70.

For **iter** and **ds** values ,respectively around 200-400 and 100, the path, even with different tests, does not change drastically: this is due to the fact that the algorithm looks for the minimum absolute path, that is the one that joins the starting and ending points with a straight line; with many configurations and neighbors available, the minimum path will tend to approach the straight line joining these two points.

Path through multiple waypoints

In order to test the control algorithms that will be presented in the next chapter, a path with multiple waypoints has been implemented:

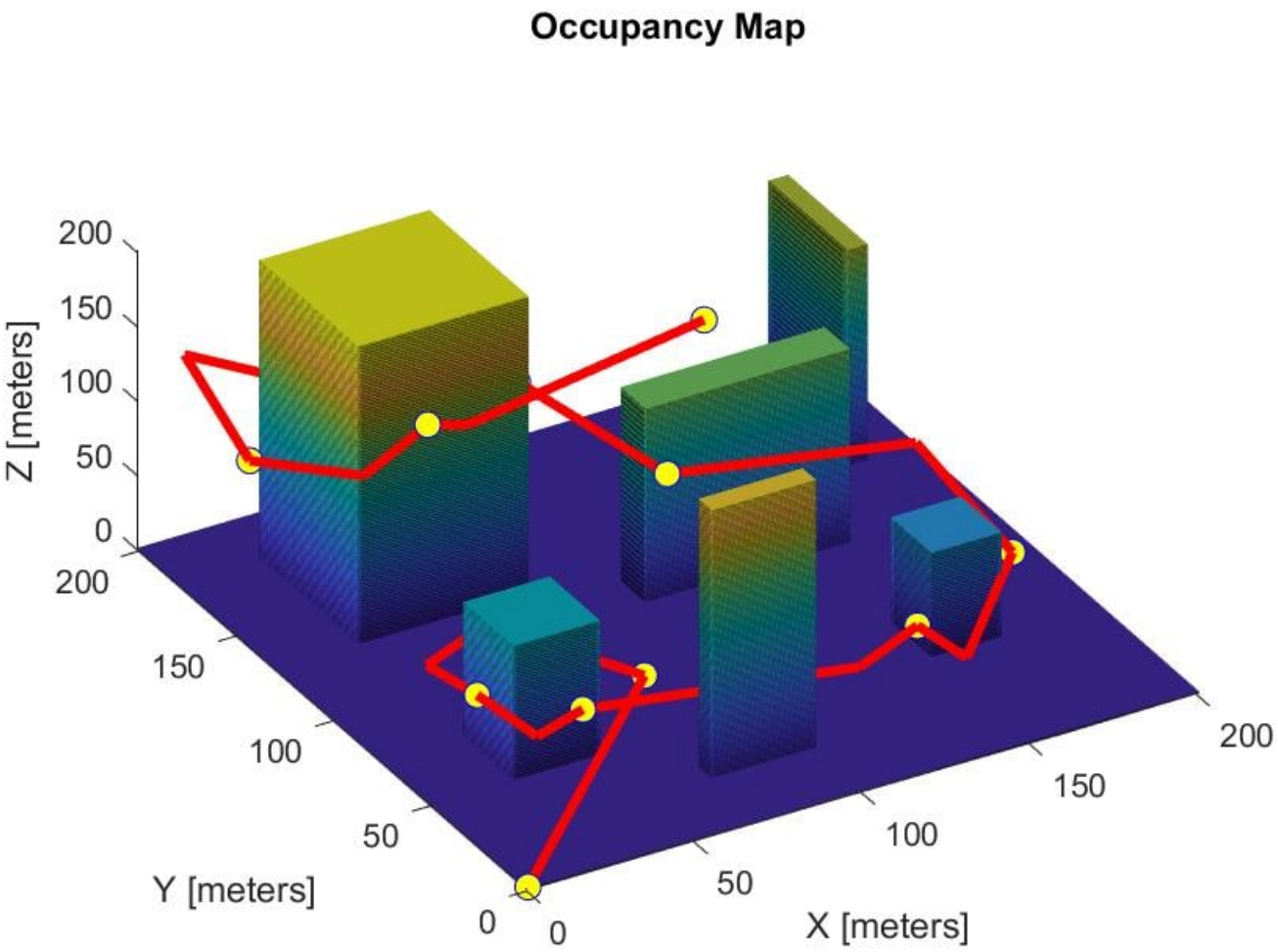


Figure 1.8: Path obtained with iter = 300 and ds = 100

The code present in **multi.waypts.m** has been implemented to obtain the plot above.

Once the waypoints have been settled, the algorithm uses the same functions presented at the beginning of the chapter, with one big difference: the A^* algorithm this time is not executed only once, but cyclically and the starting and the ending point of the path are updated as waypoints are reached.

At the end of the execution, the search algorithm calculates the following configurations that the robot must follow to reach the goal without colliding with obstacles:

$$\text{pts} = \begin{bmatrix} 1 & 1 & 1 \\ 70 & 60 & 30 \\ 40 & 90 & 50 \\ 16 & 79 & 51 \\ 20 & 60 & 50 \\ 24 & 36 & 47 \\ 40 & 40 & 50 \\ 117 & 30 & 39 \\ 140 & 40 & 40 \\ 139 & 13 & 50 \\ 180 & 60 & 40 \\ 172 & 96 & 78 \\ 100 & 100 & 100 \\ 90 & 160 & 100 \\ 91 & 192 & 109 \\ 40 & 190 & 100 \\ 5 & 185 & 146 \\ 10 & 160 & 100 \\ 18 & 115 & 136 \\ 40 & 120 & 150 \\ 52 & 120 & 142 \\ 140 & 150 & 120 \end{bmatrix}$$

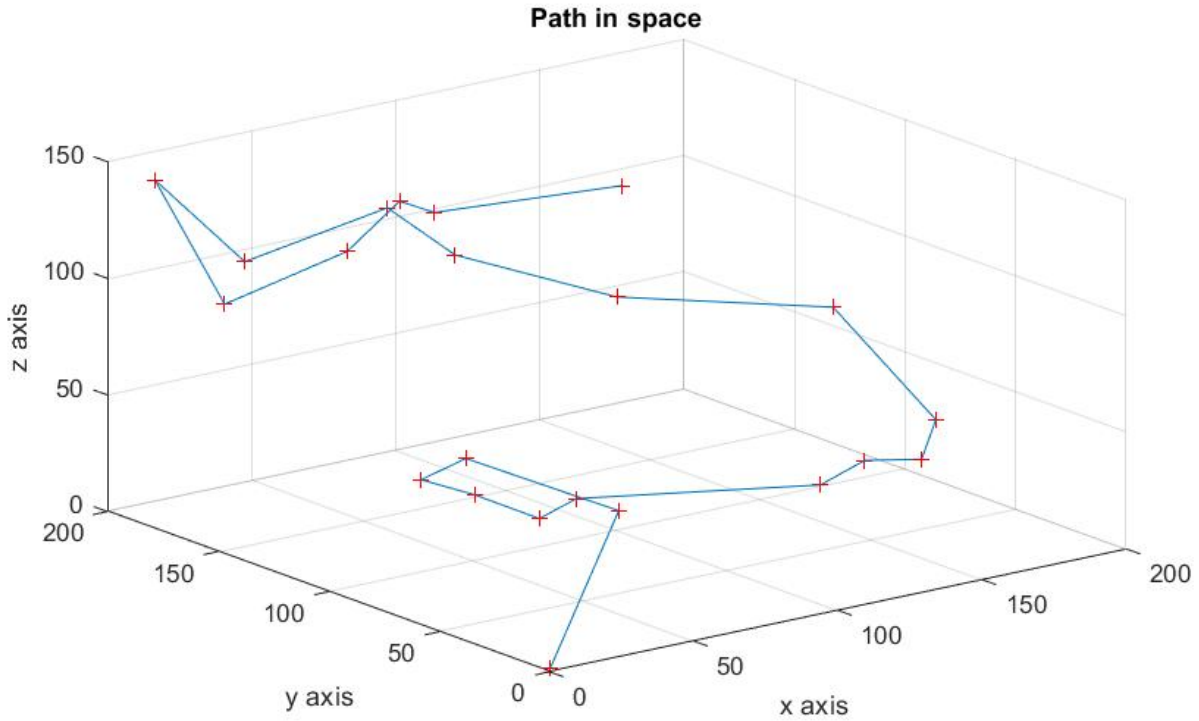
Chapter 2

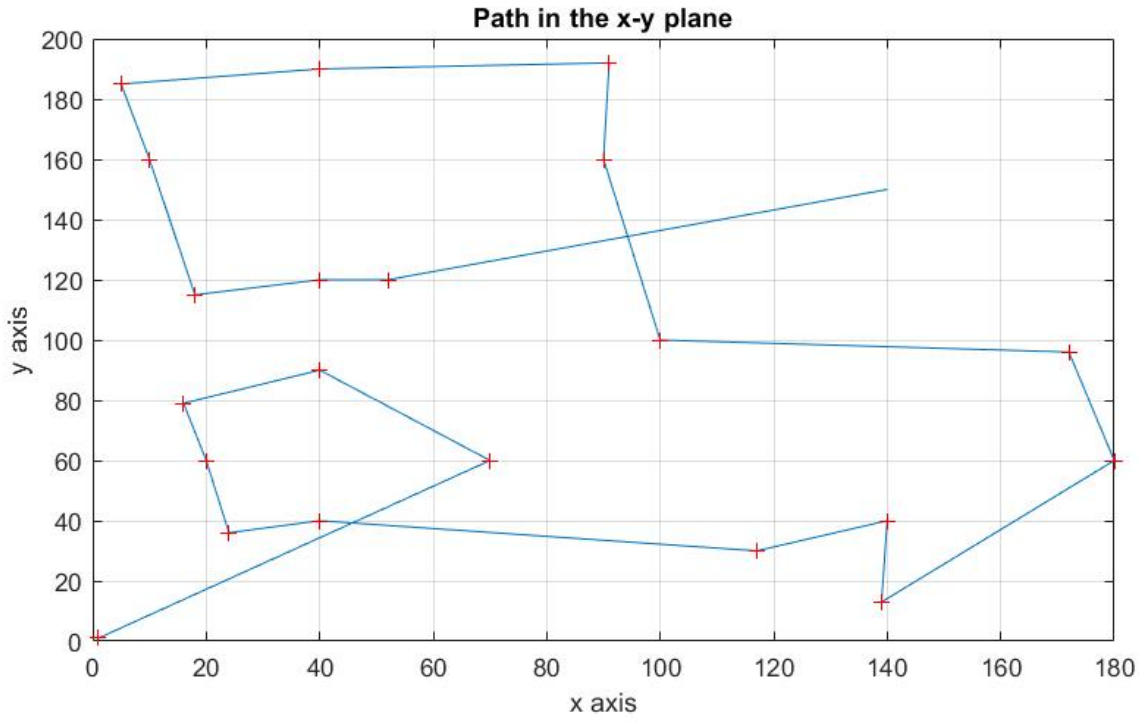
Trajectory Planning

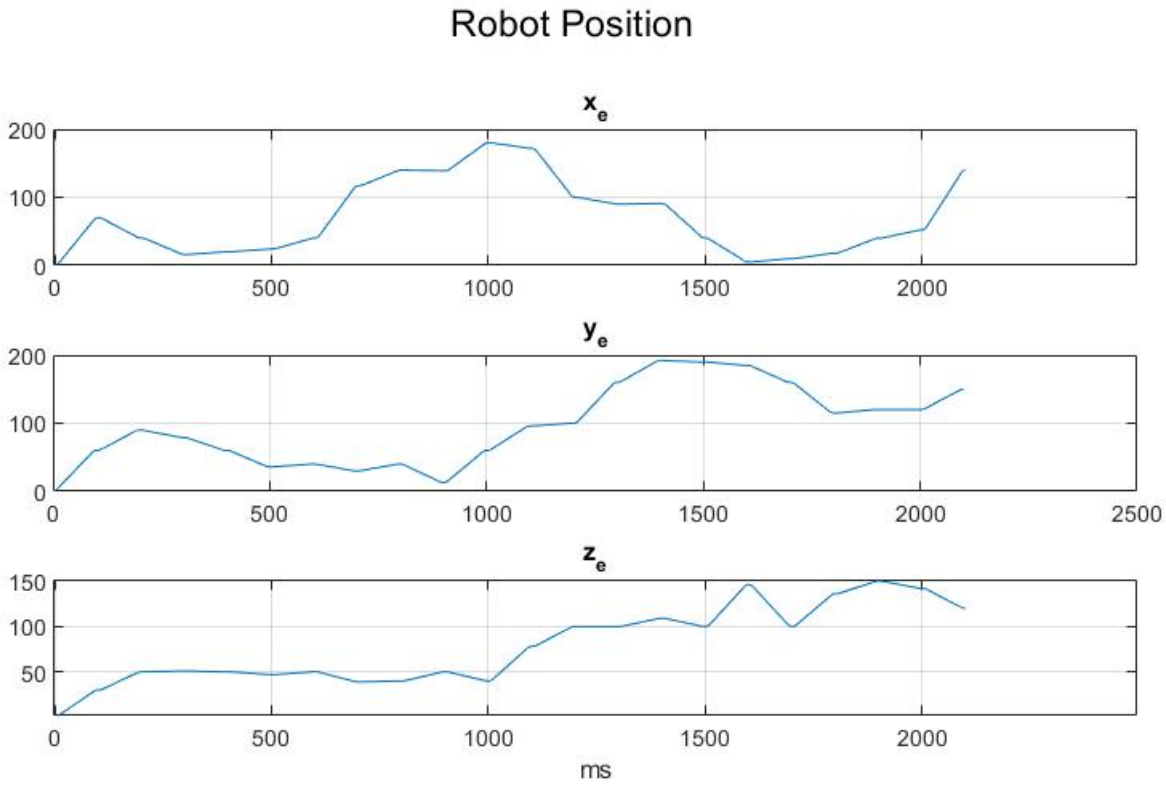
The goal of trajectory planning is to generate the reference inputs to the motion control system which ensures that the robot executes the planned trajectories. So we have to generate a time sequence of values of the desired trajectory, that is, to specify a timing law in terms of velocities and accelerations at each point.

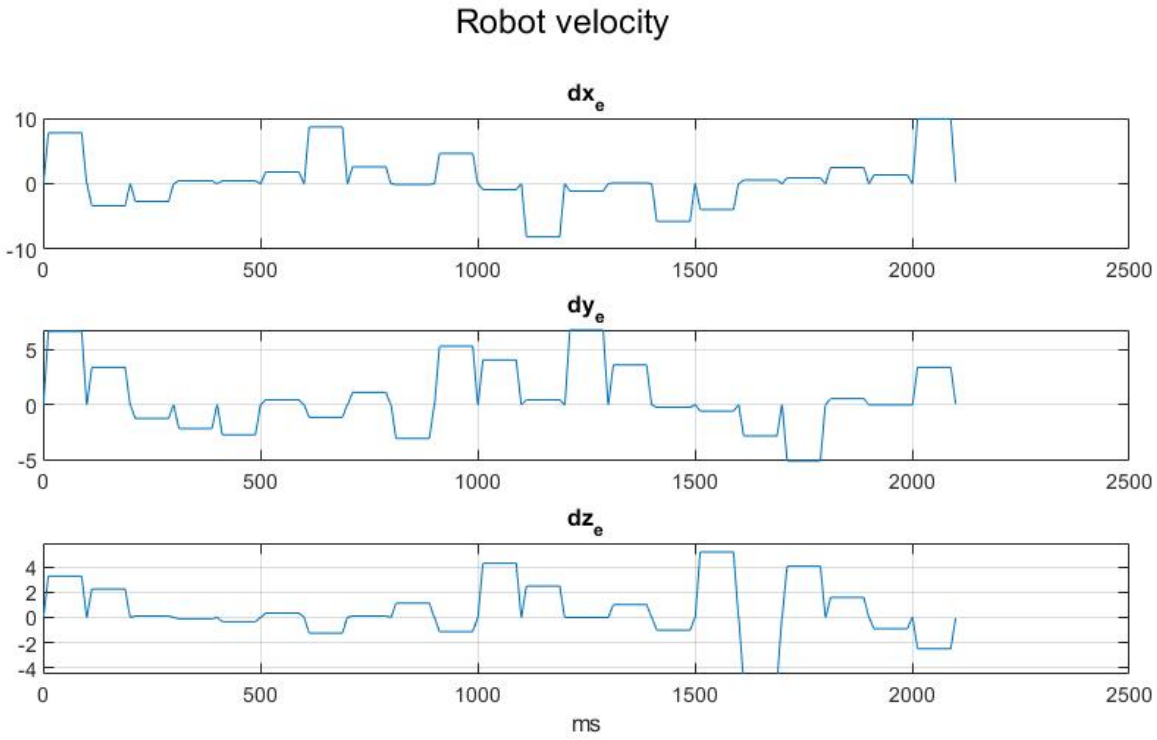
In this case, a trapezoidal velocity profile is assigned, which imposes a constant acceleration in the start phase, a cruise velocity, and a constant deceleration in the arrival phase.

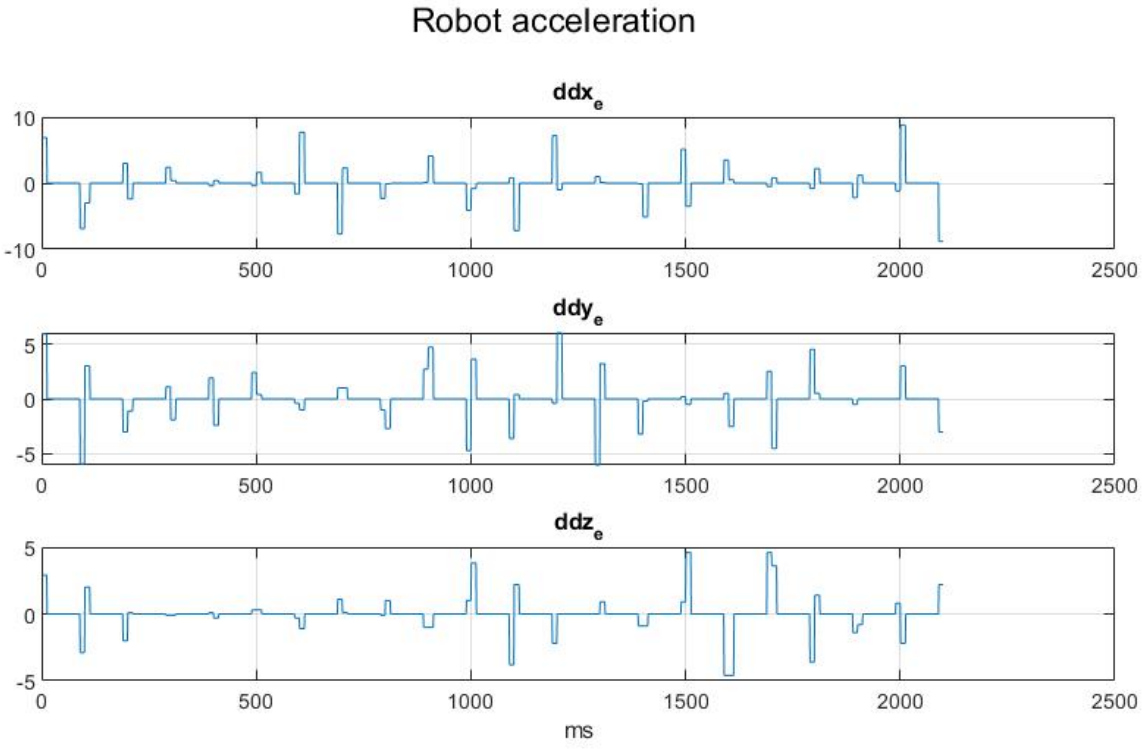
Therefore, by imposing the waypoints previously obtained, we have:




Path in the x-y plane
y axis
x axis
0
20
40
60
80
100
120
140
160
180
200


Robot Position
x_e
y_e
z_e
ms
0
500
1000
1500
2000
2500
50
100
150
200


Robot velocity
dx_e
10
0
-10
0
500
1000
1500
2000
2500
dy_e
5
0
-5
0
500
1000
1500
2000
2500
dz_e
4
2
0
-2
-4
0
500
1000
1500
2000
2500
ms


Robot acceleration
ddx_e
10
0
-10
0
500
1000
1500
2000
2500
ddy_e
5
0
-5
0
500
1000
1500
2000
2500
ddz_e
5
0
-5
0
500
1000
1500
2000
2500
ms

Chapter 3

Controllers

Passivity Based Control without Observation of external disturbances

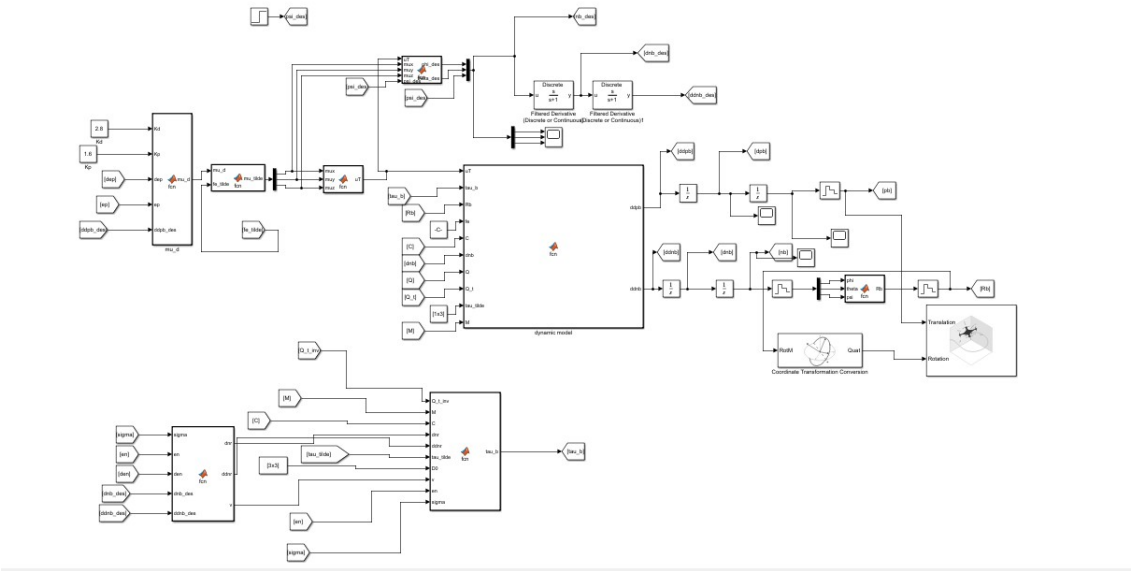


Figure 3.1: Passivity-based control scheme

This control technique is chosen for its intrinsic robustness, given by avoiding the feedback linearization related to the angular part. Starting from the RPY quadrotor Dynamic Model:

$$m\ddot{p}_b = mge_3 - u_T R_b e_3 + f_e \tag{3.1}$$

$$M\ddot{\eta}_b = -C\dot{\eta}_b + Q^T \tau^b + \tau_e \tag{3.2}$$

where m represents the quadrotor mass and it is equal to 1.2 Kg, g is the gravity acceleration and u_T, known as the total thrust produced by the propellers, is directed along z_b . It is used to compensate g and to control the vertical movement.

τ_b, on the other hand, is the torques control input and it is useful to control the orientation.

f_e and τ_e represent the external disturbances and the unmodelled dynamics which, if relevant, seriously affect the system.

Thanks to the motion planner, the following parameters are known:

$p_{b,d}, \psi_d$

because of underactuation of the system.

By knowing ψ_{des} it is possible to obtain the Euler angles ϕ_{des} and θ_{des} to complete the triad to describe the attitude. By deriving and filtering twice the velocity and acceleration are retrieved.

Defined the position and orientation errors and their first and second derivative, let consider the following expressions:

$$\dot{\eta}_r = \dot{\eta}_{b,d} - \sigma e_\eta \tag{3.3}$$

$$\ddot{\eta}_r = \ddot{\eta}_{b,d} - \sigma \dot{e}_\eta \tag{3.4}$$

$\sigma = 100$

$$\nu_\eta = \dot{e}_\eta + \sigma e_\eta \tag{3.5}$$

The control input for the orientation is evaluated as:

$$\tau^b = Q^{-T}[M\ddot{\eta}_r + C\dot{\eta}_r - \hat{\tau}_e - D_0\nu_\eta - K_0 e_\eta] \tag{3.6}$$

$$D_0 = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

$K_0 = \sigma D_0$

About u_T, it is built as:

$$\mu_d = \ddot{p}_{b,d} - \frac{1}{m}(K_p e_p + K_D \dot{e}_p) \tag{3.7}$$

$$u_T = m\sqrt{\mu_x^2 + \mu_y^2 + (\mu_z - g)^2} \tag{3.8}$$

$K_p = 1.6$
$K_d = 2.8$

The UAV quadrotor system is considered such as a mass-damper-spring model with programmable stiffness and damping for a given mass and a given inertia matrix.

Position and Attitude control through PID controllers

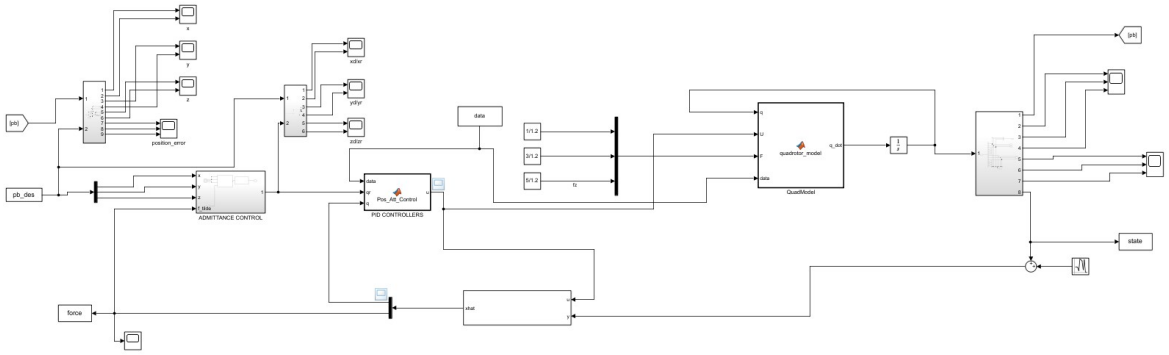


Figure 3.2: Position and Attitude control

The following control technique is improved by taking references from a Kalman filter. It returns the estimation of the state and the external forces acting on the robot. According to the case study, let consider a linearized model for the quadrotor around an equilibrium in q=0.

q represents the state vector, developed as follows:

$q = (x, y, z, \dot{x}, \dot{y}, \dot{z}, \phi, \theta, \psi, \dot{\phi}, \dot{\theta}, \dot{\psi}, \tau)$

Where τ is considered equal to : $\tau = I\dot{\omega}$

This formulation is possible according to the said NEAR-HOVER condition: $\omega = (\dot{\phi}, \dot{\theta}, \dot{\psi})$

The reformulation of the dynamic model is linked to these assumptions.

$$\dot{q} = Aq + Bu + f + w \tag{3.9}$$

$$y = Cq + Du + v \tag{3.10}$$

Moreover, the control input vector u is equal to :
$u = (u_T, \omega_c)$

Where u_T represents the thrust command and w_c is equal to
$\omega_c = (\dot{\phi}_c, \dot{\theta}_c, \dot{\psi}_c)$
and represents the three rotational body rates.

Let explain the q reference.
(x,y,z) represents the body position w.r.t. the Inertial frame, world frame .
(ϕ, θ, ψ) represents the RPY vector of Euler angles containing the orientation information of the body frame with respect to the world frame.
The dynamic model formulation is assumed in reference to the general form:

$$m\ddot{p} = -mgSe_3 - u_T e_3 \tag{3.11}$$

$$\dot{u} = \omega \tag{3.12}$$

and it is rewritten as a M-D-S System as said.

The strategy adopted to control the position and the attitude of the linearized quadrotor is based on PID-controllers. In addition, the whole system is analysed as divided in SISO subsystems.

x and θ

$$m\ddot{x} = -mg\theta \tag{3.13}$$

$$I_x\ddot{\theta} = \tau_x \tag{3.14}$$

$$\theta_r = -\frac{1}{g}[K_p(x_r - x) + K_D(\dot{x}_r - \dot{x}) + \ddot{x}_r) \tag{3.15}$$

$$\theta_c = K_\theta(\theta_r - \theta) + \dot{\theta}_r \tag{3.16}$$

z and ψ

$$m\ddot{z} = -u_T \tag{3.17}$$

$$I_z\ddot{\psi} = \tau\psi \tag{3.18}$$

$$u_T = -m[K_p(z_r - z) + k_D(\dot{z}_r - z_r)] \tag{3.19}$$

$$\dot{\psi}_c = K_\psi(\psi_r\psi) + \dot{\psi}_r \tag{3.20}$$

y and ϕ

$$m\ddot{y} = -mgSe_3 \tag{3.21}$$

$$I_y\dot{\phi} = \tau_y \tag{3.22}$$

$$\phi_r = -\frac{1}{g}[K_p(y_r - y) + K_D(\dot{y}_r - \dot{y}) + \ddot{y}_r] \tag{3.23}$$

$$\phi_c = K_\phi(y_r - y) + \dot{y}_r \tag{3.24}$$

where $K_p = [20, 20, 2]$, $K_D = [10, 10, 1]$ and $[K_\phi, K_\theta, K_\psi] = [20, 20, 1]$ represent the PID gains.

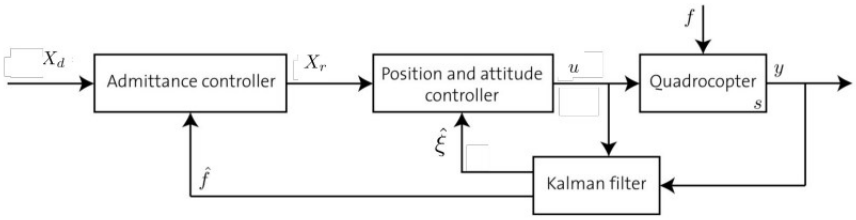


Figure 3.3: General scheme of PID controllers with Kalman Filter and Admittance Control

A Kalman Filter is involved as observer to estimate the state, useful for the control block, and the external force, which results are involved into the admittance control. The Kalman filter is built as follows:

$$\dot{q}_k = A_k q_k + B_k u_k + w_k \tag{3.25}$$

$$y_k = C_k q_k + C_k u_k \tag{3.26}$$

w_k represents the covariance matrix of the process noises and v_k represents the covariance matrix of the measurement noise.

The Kalman filter gains are evaluated through **lqe and lqr** Matlab function.

The admittance control strategy takes as input the motion planification results, in terms of desired position, and the external forces estimated by the Kalman filter. The control strategy modifies the desired trajectory through the external estimated forces which weigh on the system. The admittance control allows the successful interaction between the quadrotor and external interaction, for instance human-robot interaction. The projected control works in case of estimated forces higher than a threshold value. The admittance control scheme can be traced back to a second order system, as a mass-damper-spring system.

$$M(\ddot{p}_d - \dot{p}_r) + D(\dot{p}_d - \dot{p}_r) + K(p_d - p_r == -F \tag{3.27}$$

$$M = \begin{bmatrix} m_a & 0 & 0 \\ 0 & m_a & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.28}$$

$$D = \begin{bmatrix} 0.9 & 0 & 0 \\ 0 & 0.9 & 0 \\ 0 & 0 & 0.9 \end{bmatrix} \tag{3.29}$$

$K = D$
$m_a = 1$

The forces which act on the system are normalized w.r.t the mass. In this way they are just produced by the noise.

The admittance control gives as output a reference of position. It will be compared to the estimated state into the PID controllers block to build the control input.

The admittance control represents an useful way to apply the Kalman filter estimation results.

Simulations

For the first simulations, the external disturbances are set as constant and equal to :

$$f = (f_x, f_y, f_z) = (1, 3, 5) \text{ N}$$

For the passivity based control it is considered also τ_e as disturbance and it is equal to (0.1 , 0.1 , 1) Nm.

The following pictures show the position and attitude errors trends and the linear velocity error.

These results are about the Passivity-based Control.

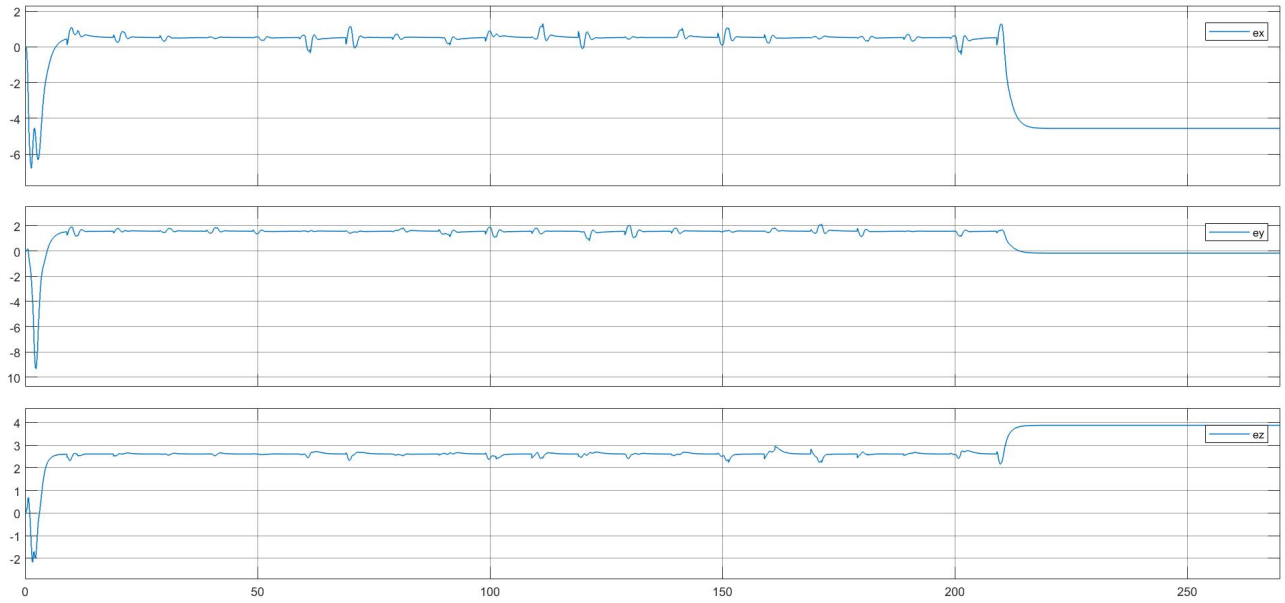


Figure 3.4: Position Error with respect to the time

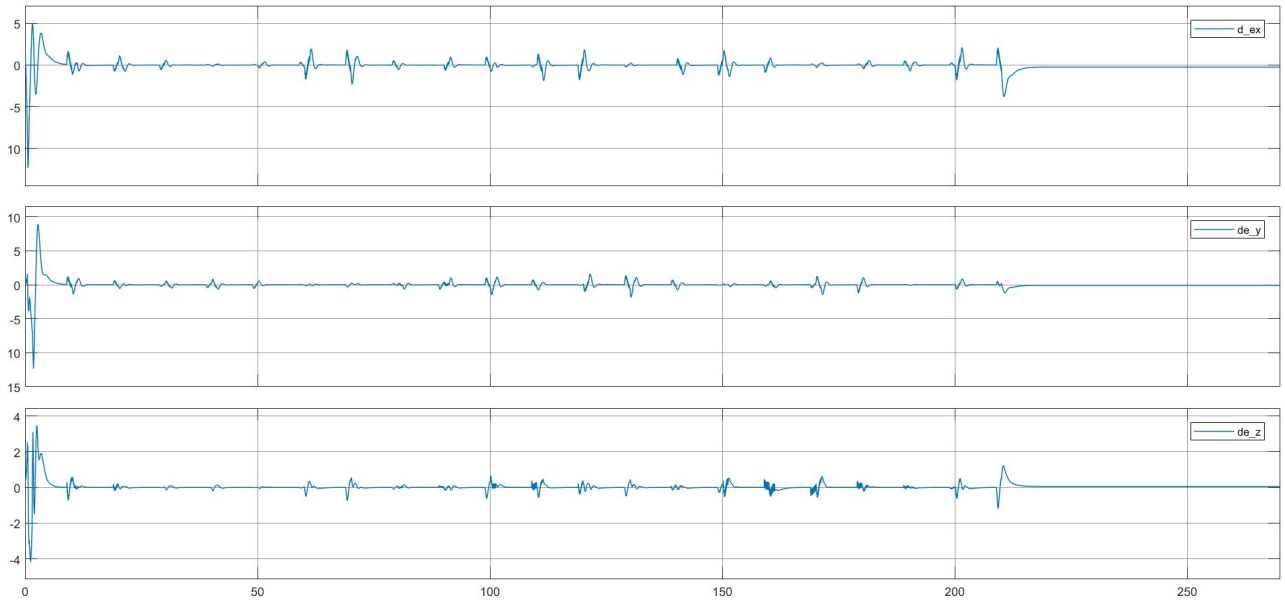


Figure 3.5: Velocity Error with respect to the time

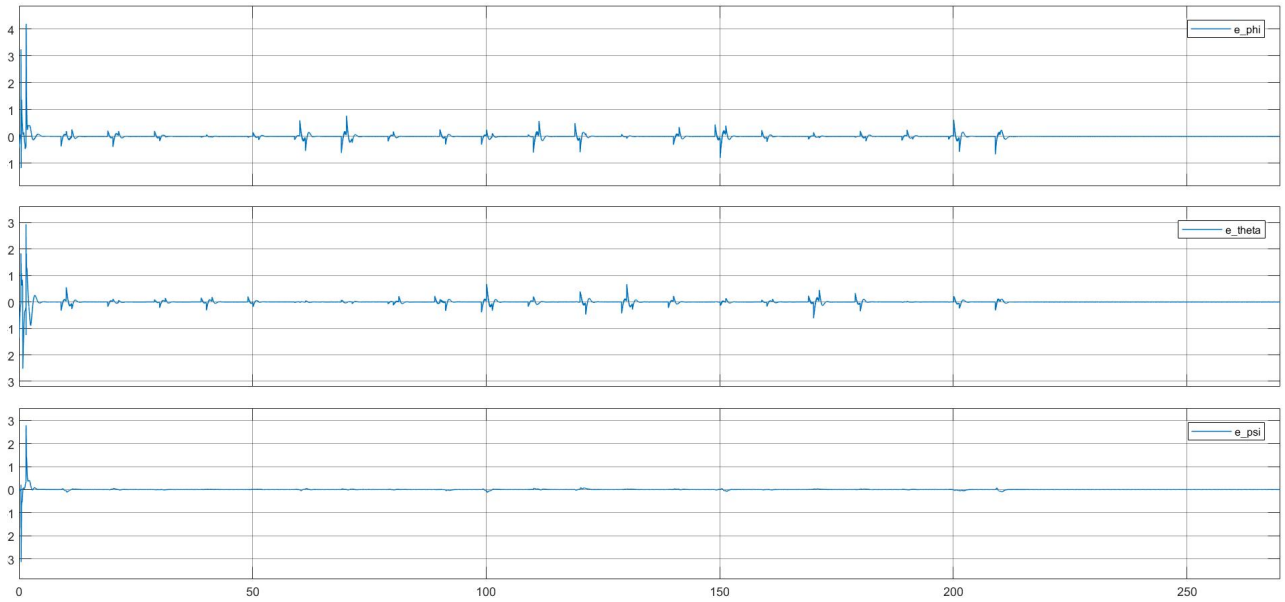


Figure 3.6: Attitude Error with respect to the time

By increasing the external disturbances and setting $f = (10, 30, 50)N$ and $\tau = (1, 1, 5)Nm$ the following results are obtained:

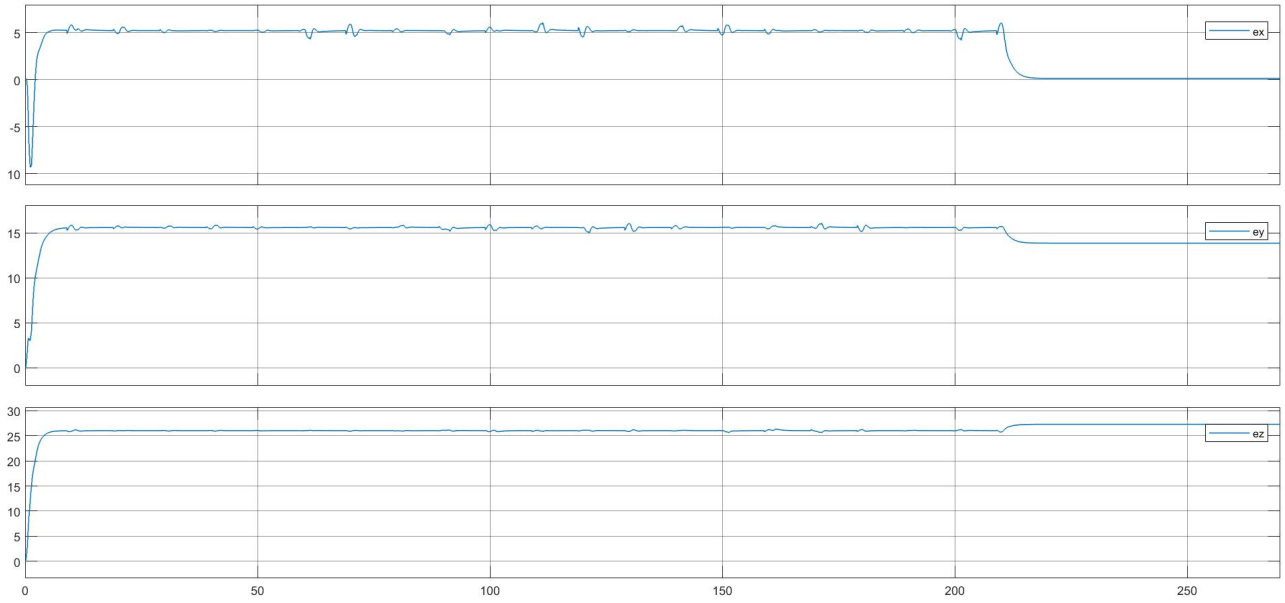


Figure 3.7: Position error with respect to the time

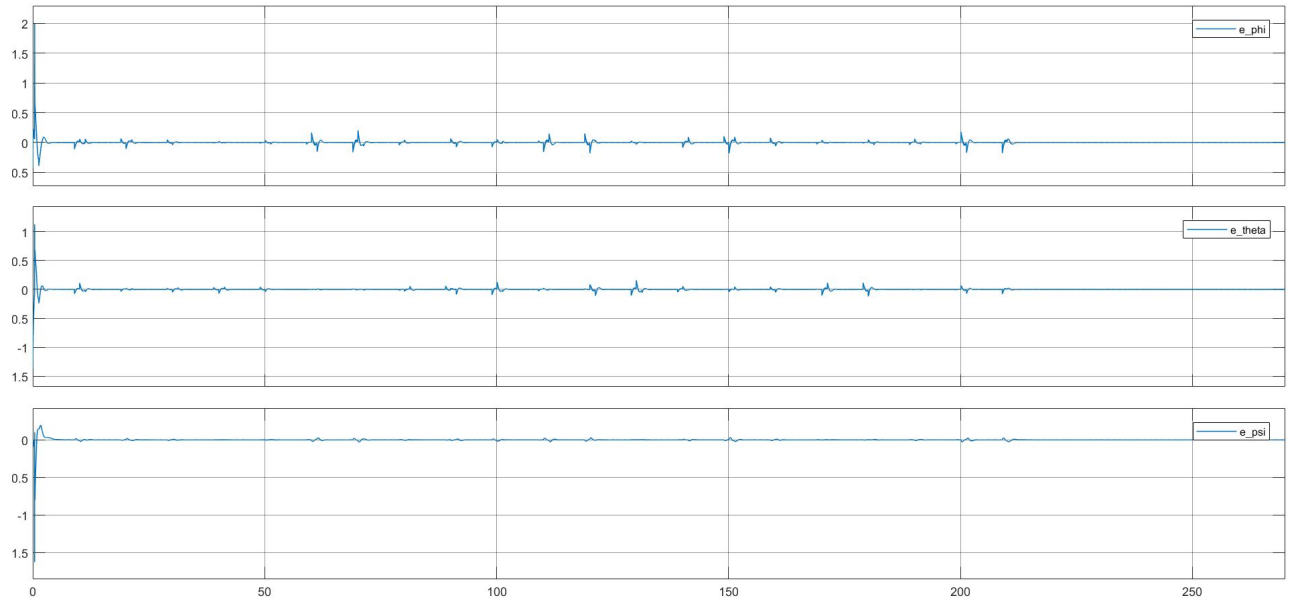


Figure 3.8: Attitude error with respect to the time

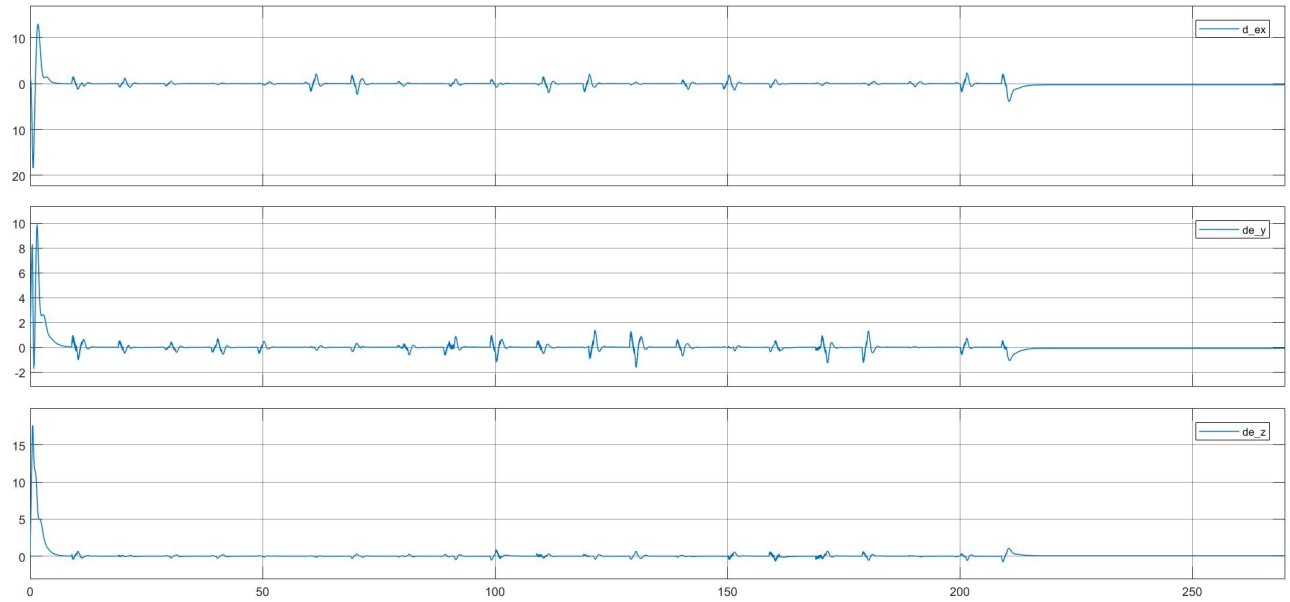


Figure 3.9: Velocity error with respect to the time

On the other hand, by refering to the first simulation disturbances for the PID controllers with Kalman Filter and admittance control, it will be show the position and orientation errors, the position reference obtained from the Admittance Control compared to the desired position and the estimated forces.

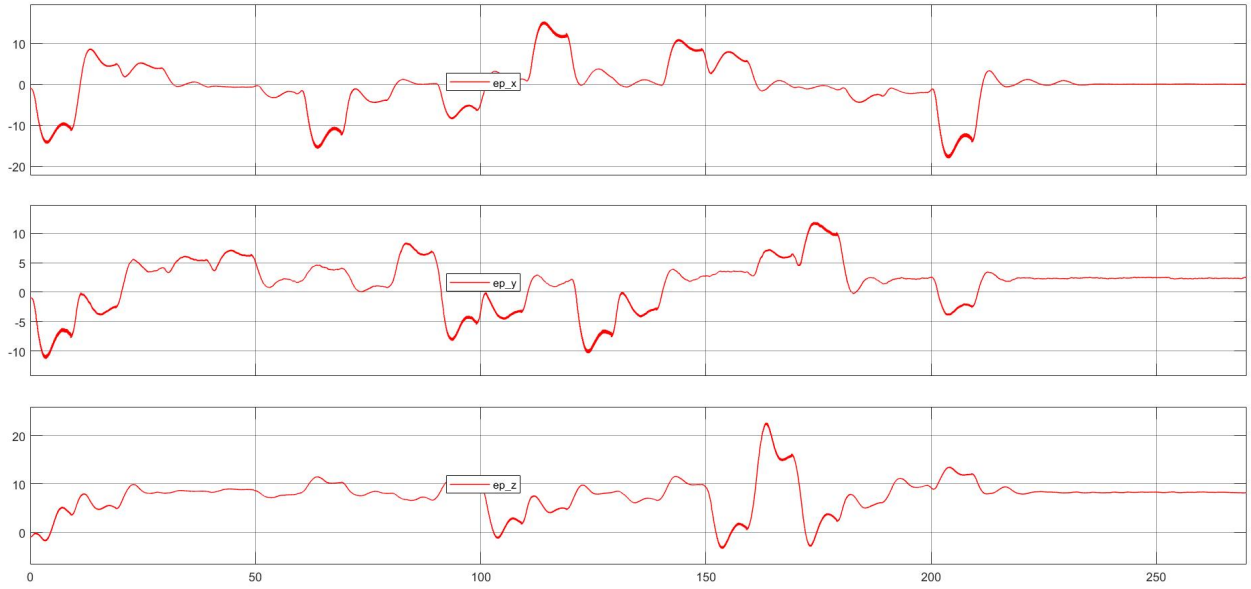


Figure 3.10: Position Error PID with respect to the time

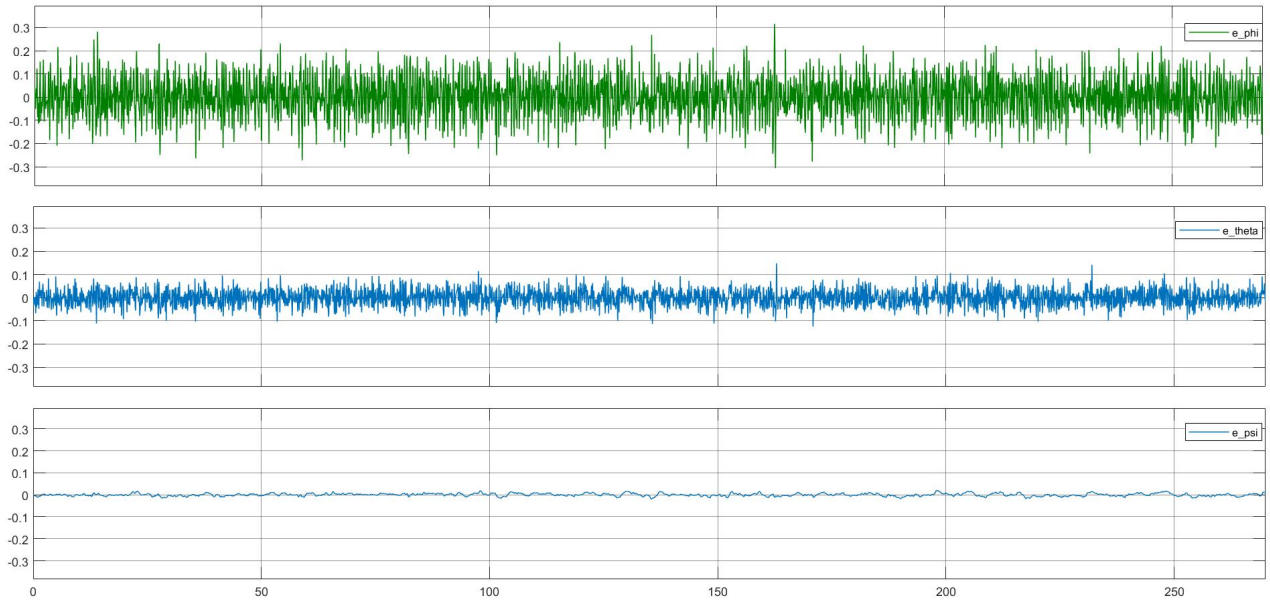


Figure 3.11: Orientation Error PID with respect to the time

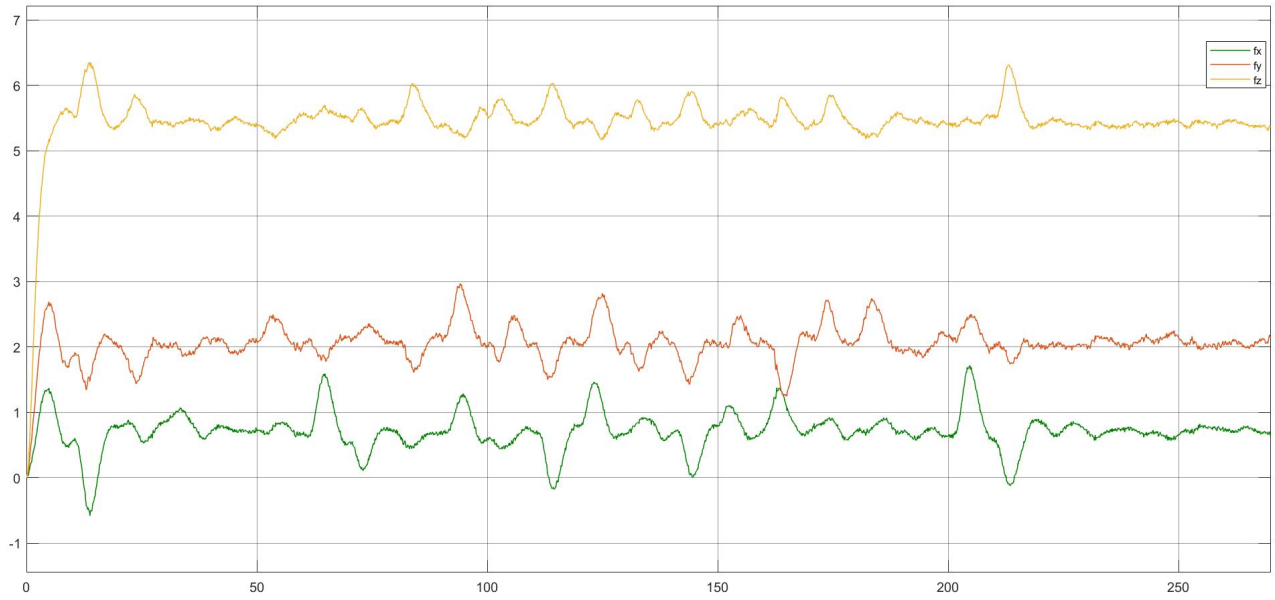


Figure 3.12: Force Estimation

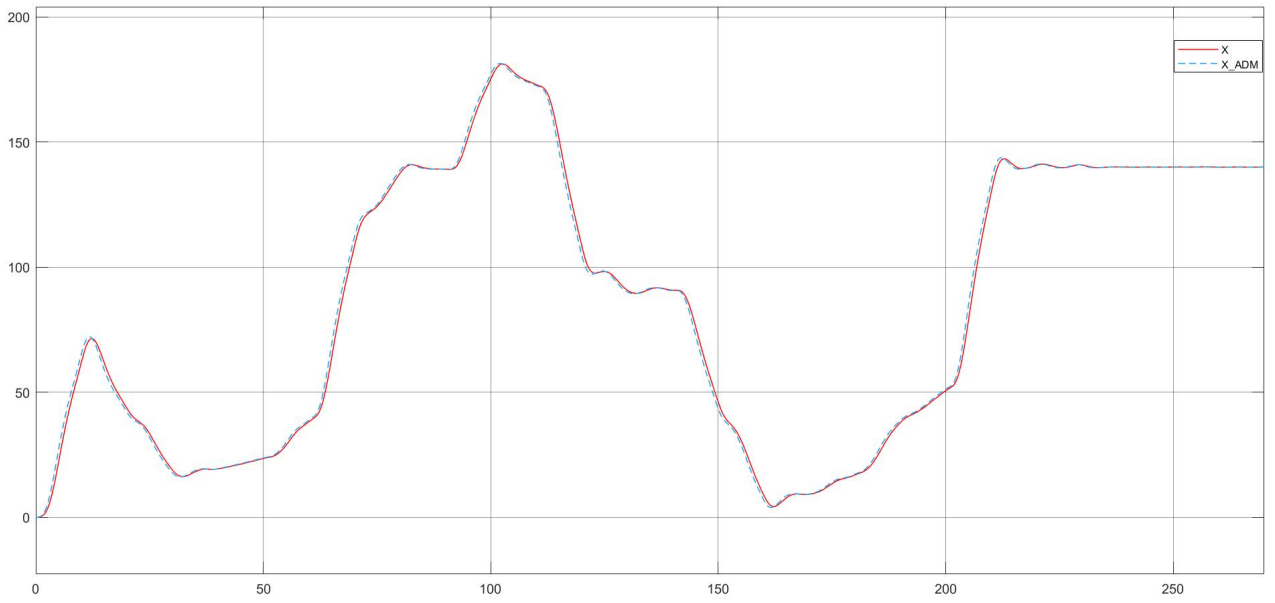


Figure 3.13: x and x-reference compared with respect to the time

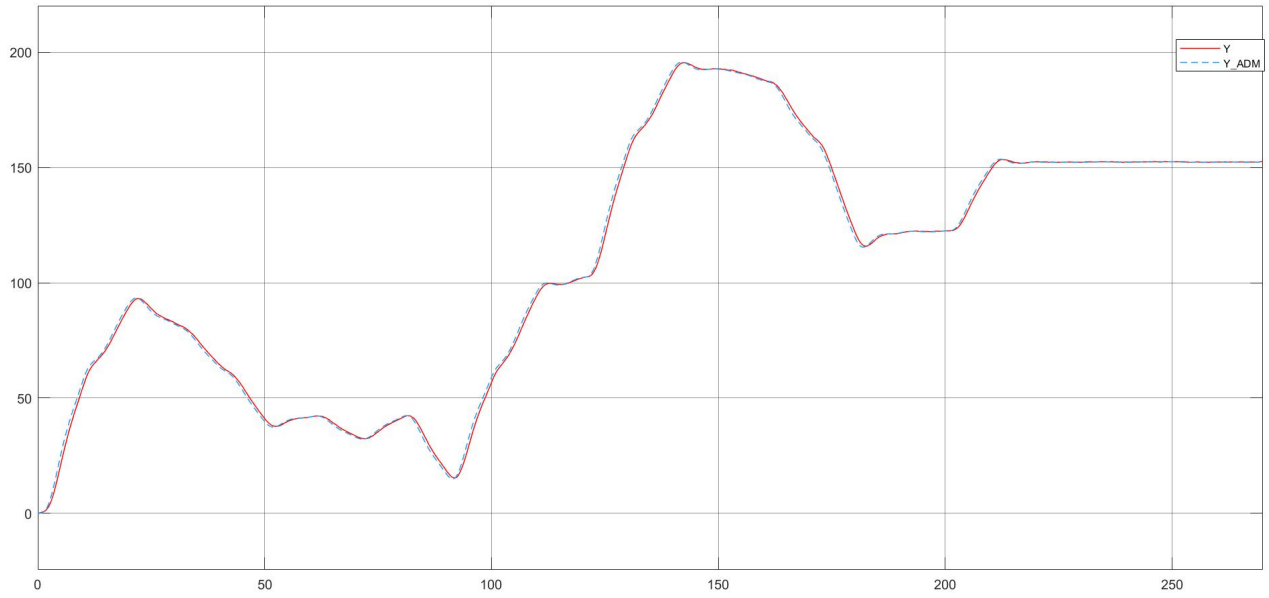


Figure 3.14: y and y-reference compared with respect to the time

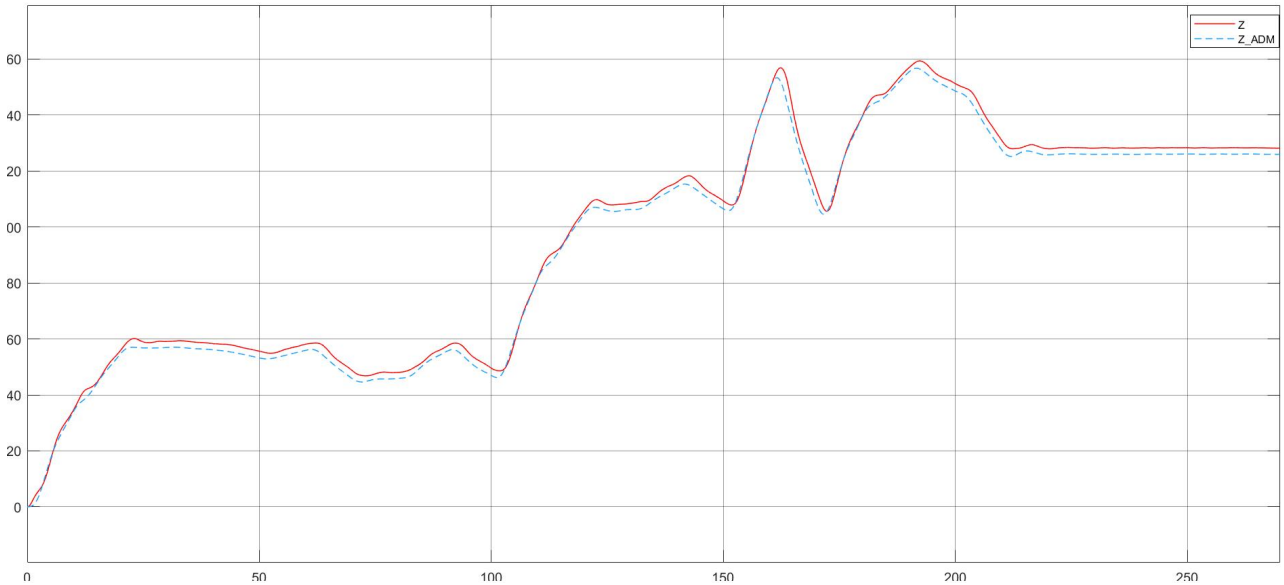


Figure 3.15: z and z-reference compared with respect to the time

By increasing the external disturbances and setting $f = (10, 30, 50)N$ the following results are obtained:

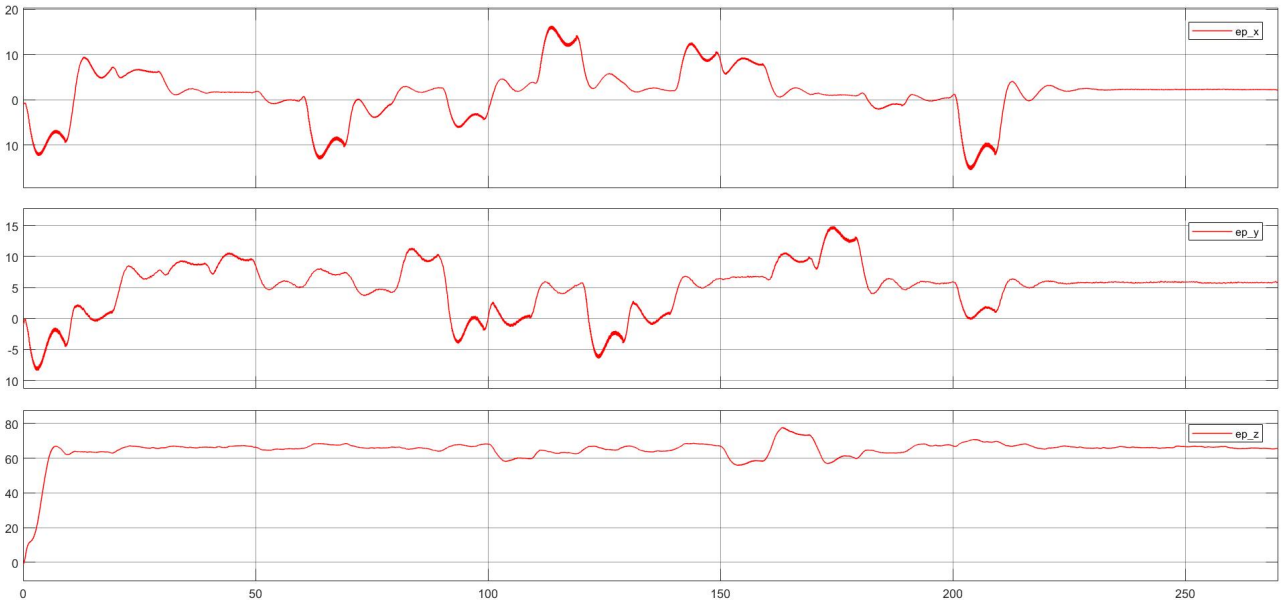


Figure 3.16: Position Error with respect to the time

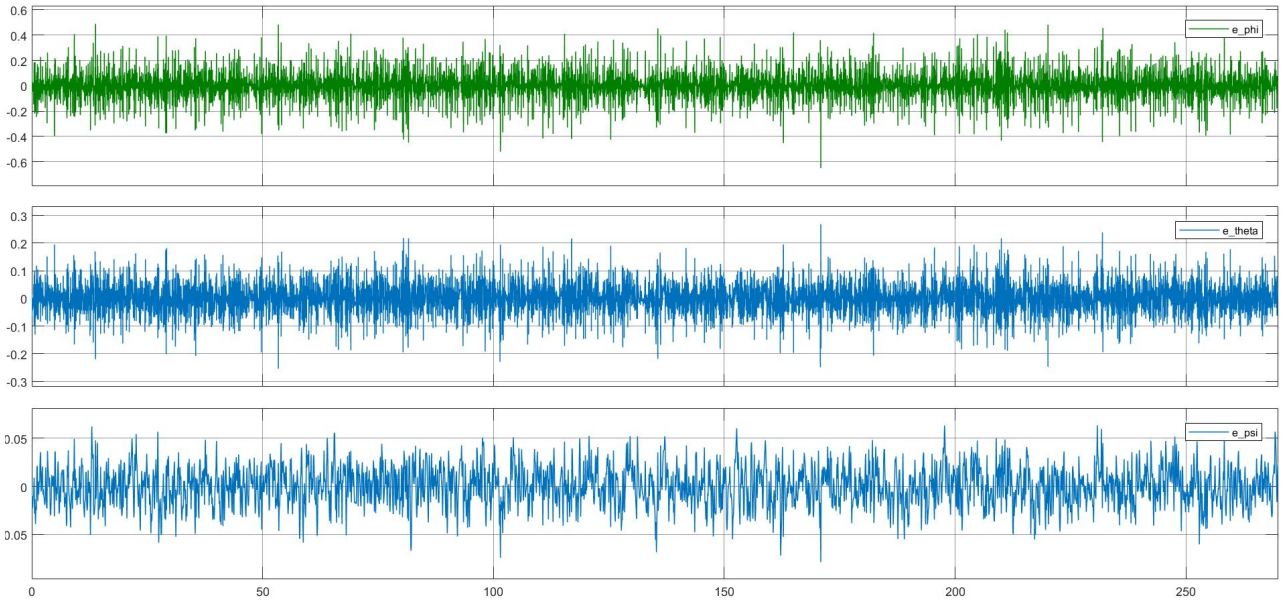


Figure 3.17: Attitude Error with respect to the time

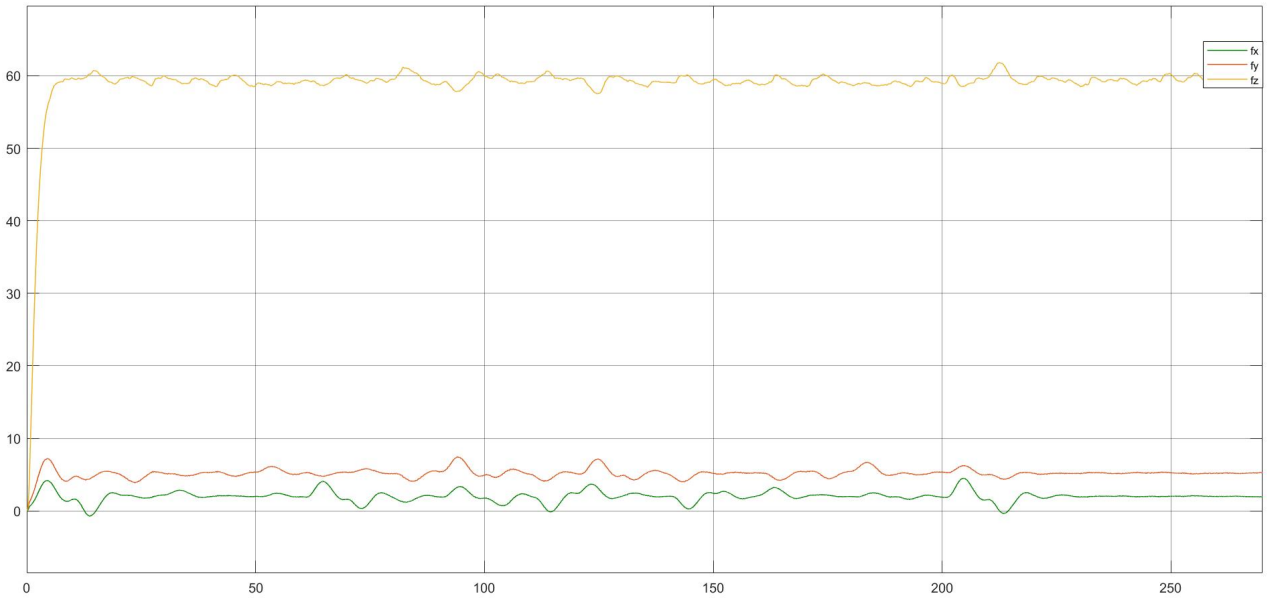


Figure 3.18: Force Estimation

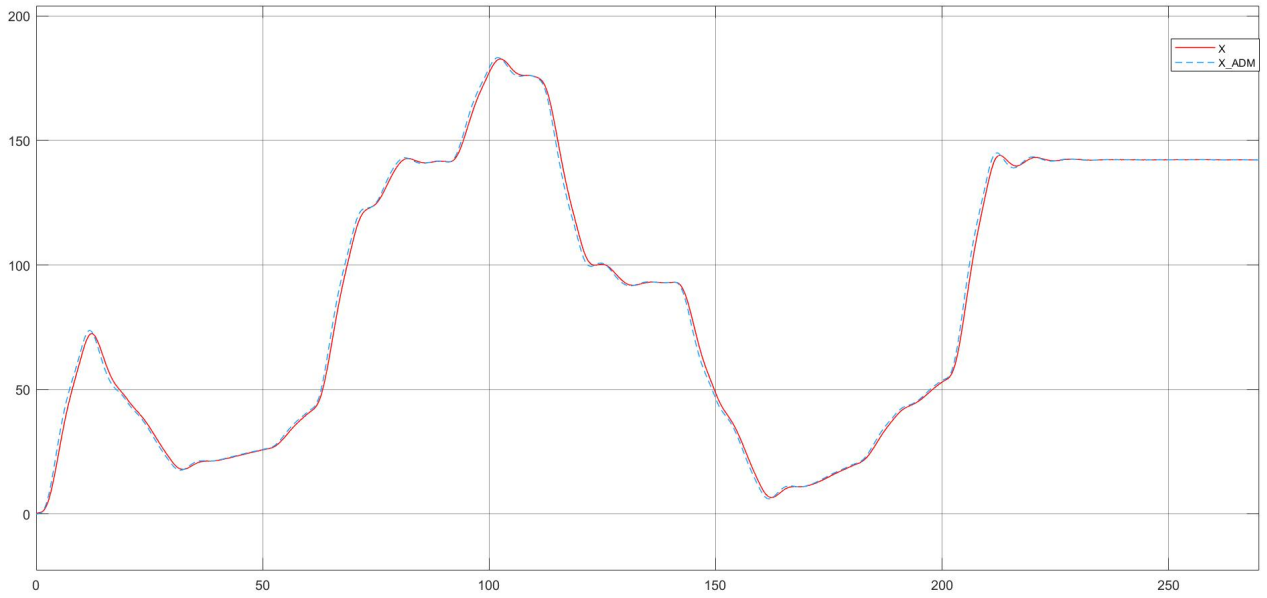


Figure 3.19: x and x-reference compared with respect to the time

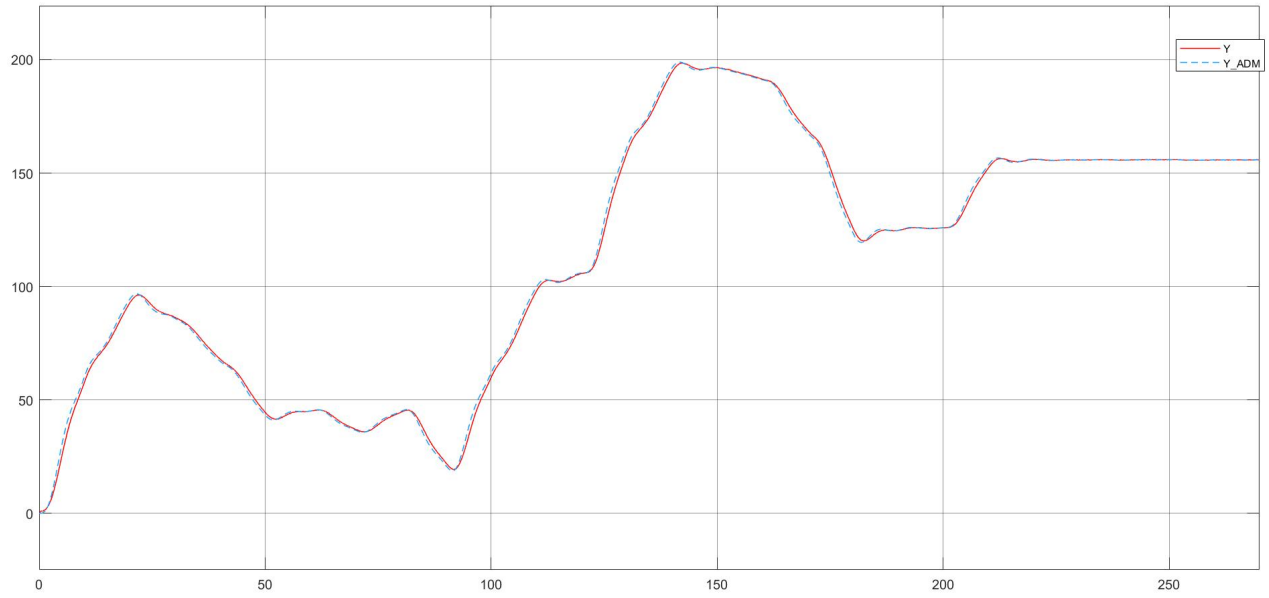


Figure 3.20: y and y-reference compared with respect to the time

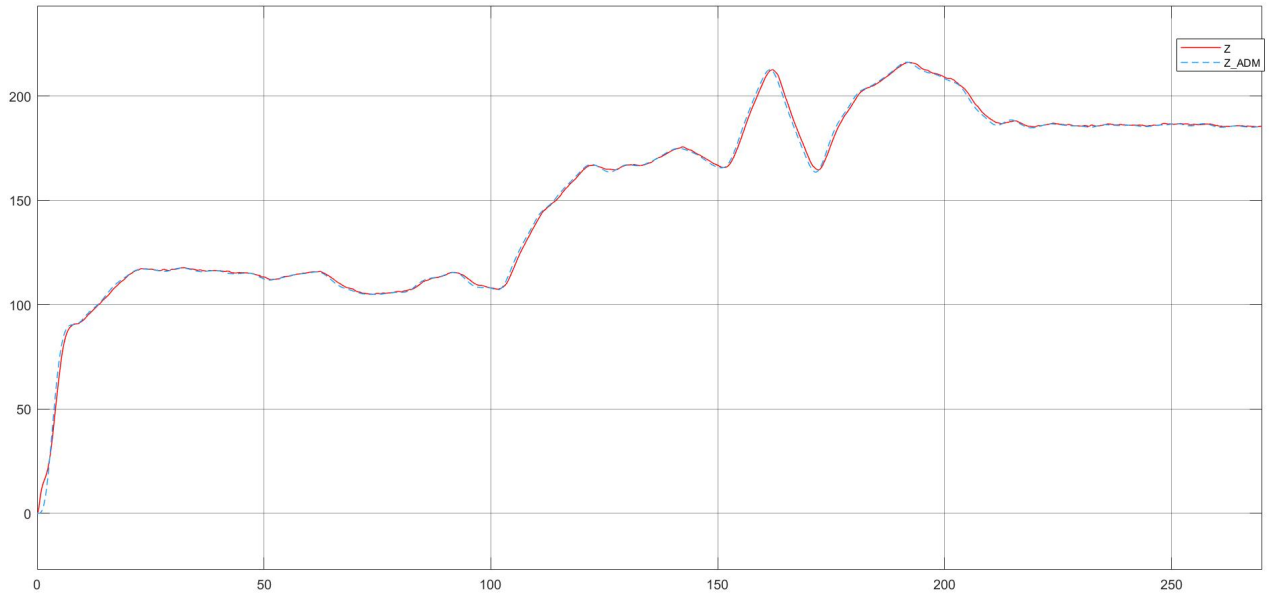


Figure 3.21: z and z-reference compared with respect to the time

Conclusion

The Passivity Based Control offers better results in terms of position error in condition of small external disturbances, but this error is constant because of the absence of an observer which allows to compensate these effects. The PID controllers, on the other hand, touches higher peak values in terms of position error, but its average is equal to zero.

By increasing the disturbances, these trends are confirmed, but for too relevant external effects the passivity based control stops to offer satisfying results. Moreover, the PID control with Kalman filter and admittance control takes in count also the linearization error.

About the orientation, the two schemes control it differently. The PID controller shows a lower attitude error than the Passivity Based one. The Passivity based control regulates the orientation through τ_b.

On the other hand, the PID controllers scheme does not directly control the orientation through τ_b, but it commands the angular velocity.

The second control technique represents an useful possibility, thanks to the Kalman filter estimation and the Admittance Control, to use the estimated forces and to regulate and control the external interaction with the robot.

About the second simulation, its results are not involved into the trajectory tracking problem because of its too high errors.

It is useful to understand the controllers behaviour when the disturbances are very relevant.

Appendix A

Matlab codes

Function occ.map.m

```
function  occ_map(omap3D,start,goal)

q_start = start;
q_final = goal;


numofobstacles=6;

global mapWidth
global mapLength
global mapHeight

mapWidth = 200;
mapLength = 200;
mapHeight = 200;


width       =   [ 25  30   5  60  50  20];
length      =   [ 25   6  40  12  50  20];
height      =   [ 90 180 145 130 200  70];
xPosition   =   [ 25  70 180 100  25 150];
yPosition   =   [ 50  25 145 111 130  50];

for i=1:1:numofobstacles
[xObstacle,yObstacle,zObstacle] = meshgrid(xPosition(i):xPosition(i)+
width(i),yPosition(i):yPosition(i)+length(i),0:height(i));

xyzObstacles = [xObstacle(:) yObstacle(:) zObstacle(:)];
```

```
setOccupancy(omap3D,xyzObstacles,1)


end
[xGround,yGround,zGround] = meshgrid(0:mapWidth,0:mapLength,0);
xyzGround = [xGround(:) yGround(:) zGround(:)];
setOccupancy(omap3D,xyzGround,1)

figure("Name","3D Occupancy Map")
fig.Color = [0 0.8 0.8];

show(omap3D)
hold on
plot3(q_start(1), q_start(2), q_start(3),'bo','MarkerFaceColor','yellow',
'MarkerSize',8)
hold on
plot3(q_final(1), q_final(2), q_final(3),'bo','MarkerFaceColor','yellow',
'MarkerSize',8)


end
```

Function q.rand.gen.m

```
function [points_rdmap] = q_rand_gen(iter,q_start, q_final,omap3D)


global mapWidth
global mapLength
global mapHeight

points_rdmap(1,:) = [q_start(1) q_start(2) q_start(3)];

 s=2;

 for i=1:1:iter

qrand = random(mapWidth, mapLength, mapHeight);
```

```
15
    if( checkOccupancy(omap3D, qrand) == -1)
17      points_rdmap(s,:) = [qrand(1) qrand(2) qrand(3)];
        s = s+1;
19   end

21  end

23  points_rdmap(s,:) = [q_final(1) q_final(2) q_final(3)];
clear length
25


27 end
```

Function seg.gen.m

```
1 function [tree] = seg_gen(ds, points_rdmap,omap3D)
ret =1;
3
indx = [];
5 indx(1,:) = [-1 -1];

7 for j = 1:1:length(points_rdmap)

9     cnd = points_rdmap(j,:);

11    for m  = 1:1:length(points_rdmap)

13        flag = 0;

15        cnd_s = points_rdmap(m,:);

17        ds_i = distance_3d(cnd,cnd_s);

19            for  v =1:1:length(indx(:,1))

21                if(indx(v,2) == j && indx(v,1) == m)
                          flag = 1;
23                end
```

```
25                end
                 if(ds_i > 0 && ds_i < ds && flag ==0 )
27
                     indx(ret,:)=[j m];
29                   ret = ret+1;

31               end


33
   end
35 end

37 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

39
   l =1 ;
41 for p = 1:1:length(indx)

43      first = points_rdmap(indx(p,1),:);
       second = points_rdmap(indx(p,2),:);
45     ok = 1;

47 pts = segm(first,second);

49 for k = 1:1:length(pts)
   point = pts(k,:);
51
       if(checkOccupancy(omap3D, point) == 1)
53         ok = 0;
           break
55
       end
57


59
   end
61
   if (ok ==1)
63     tree(l,:) = [ indx(p,1) indx(p,2)];
       l=l+1;
65
   end
67
   end
```

```
69
end
```

Function segm.m

```
function pts = segm(first, second)
2
t = 100;
4
seg = first(1) : (second(1) - first(1))/t : second(1);
6 if(length(seg) == 0)
    seg = first(1)*ones(1,t+1);
8 end

10 seg1 = first(2) : (second(2) - first(2))/t : second(2);
if(length(seg1) == 0)
12     seg1 = first(2)*ones(1,t+1);
end
14
seg2 = first(3) : (second(3) - first(3))/t : second(3);
16 if(length(seg2) == 0)
    seg2 = first(3)*ones(1,t+1);
18 end


20
for k = 1:1:length(seg)
22 pts(k,:) = [seg(k)  seg1(k)  seg2(k)];
end
24
end
```

Function A.star.m

```
function [ tree_dfs , sol_found] = A_star(points_rdmap,tree)


%%%%%%%%%%ADJ MATRIX%%%%%%%%%%%%%%%%%
ADJ = zeros(length(points_rdmap), length(points_rdmap) );

for p = 1:1:length(points_rdmap)

    for s=1:1:length(tree)
        if(tree(s,1) == p)
            ADJ(p,tree(s,2)) =distance_3d(points_rdmap(p,:) ,
            points_rdmap(tree(s,2),:));

        end

    end


end
goal = length(points_rdmap);
sol_found = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 for i =1:1:length(ADJ(1,:))
 nodes(i,:) = [i 0];
 end

OPEN  =[];
final = [];
OPEN(1,:) = [1 0 0];
nodes(1,2) = 1;

N_best = OPEN(1,:);
final(end+1,:)  = [1 0 0];
OPEN(1,:) = [];

for p =1:1:length(ADJ(1,:))
    if(ADJ(N_best(1),p) > 0)
        OPEN(end+1,:)  =  [p N_best(1)  ADJ(N_best(1),p) ];
        final(end+1,:) =  [p N_best(1)  ADJ(N_best(1),p)];
        nodes(p,2) =1;
    end
end
```

```
45
   OPEN = sortrows(OPEN,3);
47
   %%%%%%%%%%%%%%%%%%%%%%
49

51 while(1)

53 if(length(OPEN(:,1)) == 0)
       break
55 end

57 N_best = OPEN(1,:);

59
   if (N_best(1) == goal)
61     sol_found =1;
       break
63 end

65 OPEN(1,:) = [];

67 for p =1:1:length(ADJ(1,:))
       if(ADJ(N_best(1),p) > 0)
69
           if(nodes(p,2) == 0 )
71
           OPEN(end+1,:) =  [p N_best(1)  (ADJ(N_best(1),p) + N_best(3)) ];
73         final(end+1,:) = [p N_best(1)  (ADJ(N_best(1),p) + N_best(3))];
           nodes(p,2) =1;
75
           else
77
               for n=1:1:length(final)
79                 if(final(n) == p)
                       cost_p = final(n,3);
81                 end
               end
83
            flag =  N_best(3) + ADJ(N_best(1), p);
85          if(flag < cost_p )
                 for n=1:1:length(final)
87                 if(final(n) == p)
                       final(n,2) = N_best(1);
89                 end
```

```
            end
91
        end
93
    end
95 end
end
97 OPEN = sortrows(OPEN,3);
end
99

101 if(sol_found == 1)

103 ind = goal;
tree_dfs(1) = goal;
105


107

109 while (ind > 1)

111 for n=1:1:length(final)

113     if(final(n,1) == ind)

115         ind = final(n,2);
            tree_dfs(end+1) = final(n,2);
117
     end
119         end
                end
121                 end

123 tree_dfs = flip(tree_dfs);

125
end
```

Function draw.m

```
function draw(points_rdmap,vec,tree,sol_found)

if (sol_found == 1)

% for p =1:1:length(tree(:,1))
%
% f_c_1 = points_rdmap(tree(p,1),1);
% s_c_1 = points_rdmap (tree(p,1),2);
% t_c_1 = points_rdmap (tree(p,1),3);
%
% f_c_2 = points_rdmap(tree(p,2),1);
% s_c_2 = points_rdmap (tree(p,2),2);
% t_c_2 = points_rdmap (tree(p,2),3);
%
% hold on
%  plot3( f_c_1,s_c_1, t_c_1, 'x', 'Color',  [0 0.4470 0.7410])
%  hold on
%  plot3(f_c_2,s_c_2, t_c_2, 'x', 'Color',  [0 0.4470 0.7410])
%  hold on
%  line([f_c_1 , f_c_2], [s_c_1 , s_c_2], [t_c_1 , t_c_2], 'Color', 'g')
%  drawnow
% end

for p=1:1: length(vec)-1

point_one = points_rdmap(vec(p),:);
point_two = points_rdmap(vec(p+1),:);

hold on
line([point_one(1), point_two(1)], [point_one(2), point_two(2)], [point_one(3),
point_two(3)], 'Color', 'r', 'LineWidth', 3)
drawnow

end

else
    display('Solution␣not␣found!')
    end
        end
```

Function distance.3d.m

```
function d = distance_3d(q1,q2)
    d = sqrt((q1(1)-q2(1))^2 + (q1(2)-q2(2))^2 + (q1(3)-q2(3))^2);
end
```

Function random.m

```
function qrand = random(x_map,y_map,z_map)

    qrand=[floor(rand*x_map), floor(rand*y_map) , floor(rand*z_map)];

end
```

Function multi.waypts.m

```
clear all
close all
clc

%%%%Waypoints%%%%%%%%%%%%%%%

path(1,:)  =  [  1    1    1];
path(2,:)  =  [ 70   60   30];
path(3,:)  =  [ 40   90   50];
path(4,:)  =  [ 20   60   50];
path(5,:)  =  [ 40   40   50];
path(6,:)  =  [ 140  40   40];
path(7,:)  =  [180   60   40];
path(8,:)  =  [100  100  100];
path(9,:)  =  [ 90  160  100];
path(10,:)  =  [ 40  190  100];
path(11,:) =  [ 10  160  100];
```

```
path(12,:) =  [ 40  120  150];
path(13,:) =  [140  150  120];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sol_found = 0;
q_start = path(1,:);
q_final = path(13,:);

omap3D = occupancyMap3D;
occ_map(omap3D,q_start, q_final);

for p =2:1:length(path)-1
    pt = path(p,:);
 plot3(pt(1), pt(2), pt(3),'bo','MarkerFaceColor','yellow','MarkerSize',8)
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

iter  = 300;
ds    = 100 ;
vec_f = [q_start];

points_rdmap = q_rand_gen(iter,q_start,  q_final,omap3D);

for k =1:1:length(path)-1

sol_found = 0;

points_rdmap(1,:)   = path(k,:);
points_rdmap(end,:) = path(k+1,:);

tree = seg_gen(ds,points_rdmap,omap3D);

[vec ,  sol_found] = A_star(points_rdmap,tree);

if(sol_found ==1)

for  i =1:1:length(vec)
tmp(i,:) = points_rdmap(vec(i),:);
end

if(vec_f(end,:) == tmp(1,:))
    tmp(1,:) = [];
end
```

```
63  vec_f = [vec_f;tmp];

65  end


67
    draw(points_rdmap,vec,tree,sol_found);
69
    if(sol_found == 0)
71      close all
        clear final_points
73      break
    end
75
    end
```