Grenoble INP, Université Grenoble Alpes
Ecole d'ingénieurs en systèmes intelligents cybersécurisés

PX505

BioSwarm
Innovation Project

Students
PÎNDICHI Elena
JAROUSSEAU Arthur
LOPEZ Oscar

Advisor
Prof. dr. ing. PRODAN Ionela

Valence, 2024

Contents

1. Introduction

Autonomous TurtleBots are compact, versatile robots designed to navigate environments independently while avoiding obstacles and creating detailed maps. Equipped with advanced sensors such as LiDAR, cameras, and ultrasonic sensors, these robots detect and respond to their surroundings in real-time. Using algorithms like Simultaneous Localization and Mapping (SLAM), TurtleBots generate accurate representations of a room's layout, which can be used for navigation and environmental analysis. Their ability to adapt to various scenarios makes them ideal for applications in research, education, and industries requiring dynamic spatial awareness and autonomous operation.

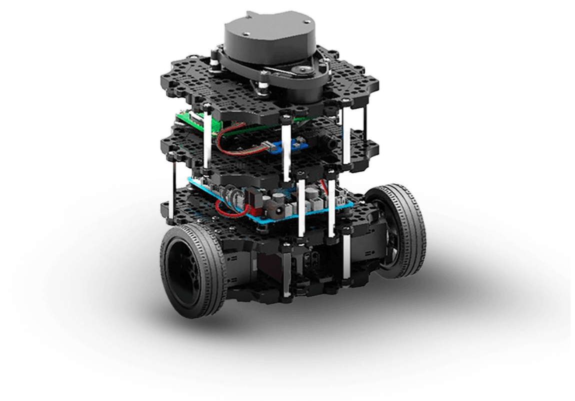


Figure 1.1.: TurtleBot

We implemented the Model Predictive Control (MPC) algorithm that enables precise obstacle avoidance and reference trajectory tracking by predicting the robot's future states and optimizing its movements accordingly. This implementation was carried out using MATLAB for algorithm design and ROS 2 (Robot Operating System) for real-time control and communication with the hardware. The integration of MPC with MATLAB and ROS 2 allows the turtlebots to efficiently navigate through dynamic environments.

2. Model Predictive Control

We have implemented the control algorithm using MATLAB and performed multiple tests, so that we can ensure its stability and performance.

1. Mathematical model

The dynamics of the robot and the vector command vectors are:

$$\zeta = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \qquad \dot{\zeta} = \begin{bmatrix} V\cos\theta \\ V\sin\theta \\ \omega \end{bmatrix} \qquad u = \begin{bmatrix} V \\ \omega \end{bmatrix} \tag{2.1}$$

Here, V stands for linear velocity and ω stands for angular velocity. This is how the system behaves or reacts under different conditions or inputs. This is a nonlinear function and hence we had to implement a Nonlinear MPC to predict its future behavior.

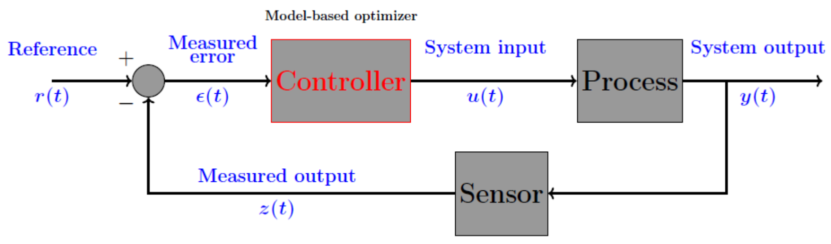


Figure 2.1.: Feedback Loop

2. Model Predictive Control Algorithm

This algorithm uses the model of the system to predict its future behavior and make optimal control decisions based on these predictions. In order to determine the future actions that the robot has to take, we have to look into a set of period of time ahead for which we make predictions about the state of the robot. This future time period is called the *prediction horizon*. We will choose the most desirable control action to input to the robot. The optimal control is the one that best achieves its goals for the system while adhering to certain given constraints.

We can estimate future states from current state as it follows, where we would introduce the *sampling time* period because a continuous based system provides outputs at every moment, while the discrete system provides them at (usually) equally divided samples of time:

$$x(k+1) = x(k) + T_s * V(k)\cos\theta(k) \tag{2.2}$$
$$y(k+1) = y(k) + T_s * V(k)\sin\theta(k) \tag{2.3}$$
$$\theta(k+1) = \theta(k) + T_s * \omega(k) \tag{2.4}$$

Each future predicted state can be computed as:

$$x(k+1) = x(k) + T_s * f_{dynamics}(x(k), u(k)) \tag{2.5}$$

Where k represents one time step as $t = kT_s$ in the control horizon.

The predictive control feedback law is computed by minimizing a predicted performance cost, which is defined in terms of the predicted sequences $\mathbf{u}$, $\mathbf{x}$. The predicted cost has the general form:

$$J(x_k, \mathbf{u}_k) = \sum_{i=0}^{N} \left(||x_{i|k}||_Q^2 + ||u_{i|k}||_R^2 \right), \tag{2.6}$$

The optimal control sequence for the problem of minimizing the predicted cost is denoted $\mathbf{u}_N^*(x_k)$ and the optimal value of the cost is $J^*(x_k) = J(x_k, \mathbf{u}_k^*)$ which is often written as J_k^*. We can rewrite the optimization cost function as:

$$\mathbf{u}_N^\star = \arg\min_{u_N} x_N^\top S x_N + \sum_{k=0}^{N-1} \left(x_k^\top Q x_k + u_k^\top R u_k \right), \tag{2.7}$$
$$\text{s.t.} \qquad x_{k+1} = x_k + T_s * f(x_k, u_k), \tag{2.8}$$
$$x_{min} \le x_{k+1} - x_k \le x_{max}, \tag{2.9}$$
$$u_{min} \le u_k \le u_{max}. \tag{2.10}$$

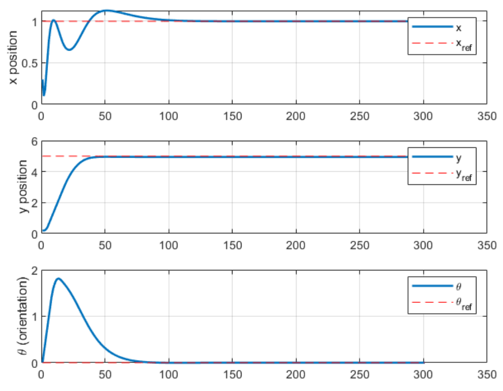
3. MATLAB implementation

Firstly, we implemented the MPC controller with reference tracking, where the reference would be a point. After multiple tests, we have chosen the best weight matrices that should be semi-positive defined, and hence we added values on the main diagonal. The weight matrices are:

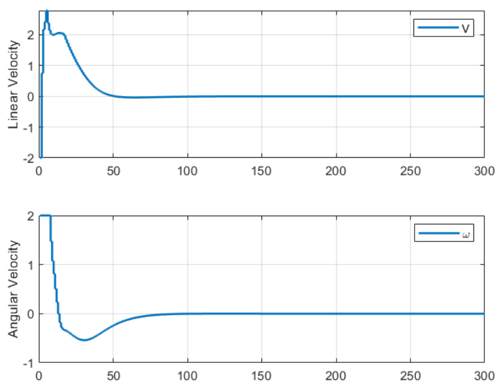
$$Q = \begin{bmatrix} 100 & 0 & 0 \\ 0 & 100 & 0 \\ 0 & 0 & 0.0001 \end{bmatrix} R = \begin{bmatrix} 1000 & 0 \\ 0 & 1000 \end{bmatrix} \qquad P = \begin{bmatrix} 1000 & 0 & 0 \\ 0 & 1000 & 0 \\ 0 & 0 & 1000 \end{bmatrix} \tag{2.11}$$

The computation time to solve the optimization problem is 0.08 seconds for a prediction horizon of 15 and a simulation horizon of 300. However, reproducing the same tests in Python did not give us the same accuracy, and we have changed the prediction horizon to 25. The reference for the command will be 0 because we do not want the control input to influence the robot.

We have improved our algorithm by creating a complete reference to track. For further tests, we tried to make the robot follow a sin and cos reference so that we ensure that it follows the trajectory. We used the matrix R as the identity matrix in MATLAB, but when we introduced this algorithm in Python, we had to change to the values from above.



(a) States tracking



(b) Inputs tracking

Figure 2.2.: Point tracking

As expected, the main problem encountered while working with the MPC algorithm is the computation time, which in this case is approximately 60 seconds. The results are almost accurate, and the states could follow the trajectory of sin and cos. We hae tested the controller on the TurtleBot and we could see that it followed the same trajectory in real-time.

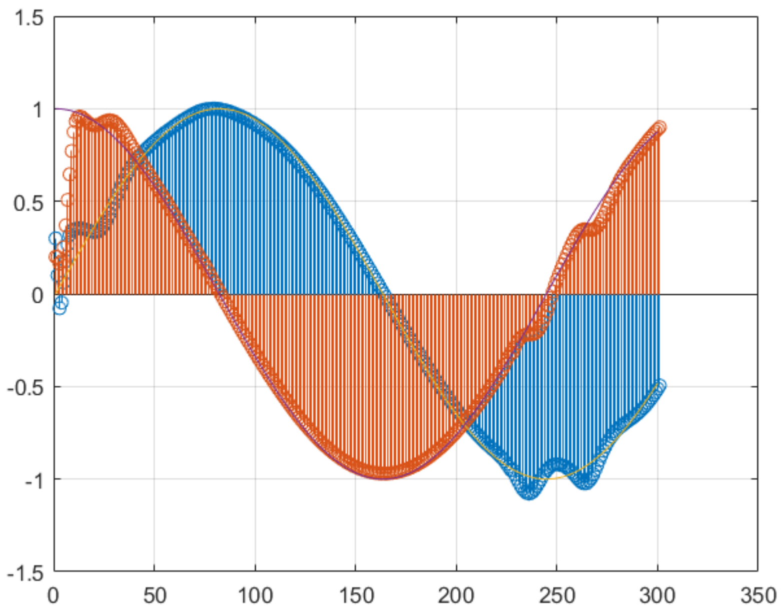


Figure 2.3.: X and Y trajectory tracking

3. ROS2 environment and implementation

We had to get used to working with the ROS2 Humble environment and prepare our setup to work with the robot.

1. Connecting with the TurtleBot

We installed the ROS2 Humble for Linux distribution 22.04 and we created our own workspace where we could add our nodes. For the installations, we followed the tutorial on ROS, and hence we got in depth with the operating system.

In our setup, we utilized a Virtual Machine (VM) and connected to the TurtleBot via SSH. To enable network connectivity, configure the VM's network as NAT (Network Address Translation). This setup allows the VM to access external networks by routing traffic through the host machine's network interface. On the host machine, create a hotspot using the name and password specified in the TurtleBot's network configuration file located at /etc/netplan/50-cloud-init.yaml. For our configuration, the credentials are: name = ubuntu and password = turtlebot. Establish an SSH session to the TurtleBot using the command: ssh ubuntu@<ip_of_turtlebot>. The TurtleBot's IP address can either be fixed or dynamically assigned and can be identified through the hotspot connections.

2. Reference tracking

2.1. Creating the map

As mentioned previously, we have implemented the MPC controller with reference tracking. The trajectory was created based on a world map of the TurtleBot from gazebo environment. The map would be created thorough SLAM algorithm. In order to create and save the map, the robot had to be moved through every corner of it and we could see the dimensions and obstacles of the map in the RVIZ application. The commands to map the world and to save it are:

```
# In the first terminal, launch the world that you want to map, for
   example, we will use the world launch:
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py

# In the 2nd terminal start the Navigation 2 stack. Note: if you are
   using Gazebo, add "use_sim_time:=True" to use the Gazebo time. If
   using the real robot, skip this argument.
$ ros2 launch nav2_bringup navigation_launch.py use_sim_time:=True

# In the 3rd terminal, start the slam_toolbox. Note: same as previously,
    if using Gazebo, add the argument to use the simulation time.
$ ros2 launch slam_toolbox online_async_launch.py use_sim_time:=True

# In the 4th terminal strat RViz:
```

```
$ ros2 run rviz2 rviz2 -d /opt/ros/humble/share/nav2_bringup/rviz/
    nav2_default_view.rviz
# To make the robot run, use the teleop command in a 5th terminal

# To save the map, you can use:
$ ros2 run nav2_map_server map_saver_cli -f my_map
```

After saving the map, we refined its contours to make it compatible with the OpenCV library in Python. Using the GIMP application, we adjusted the image and defined polytopes for the obstacles. This allowed us to generate a trajectory that avoided these obstacles. We then implemented the Dijkstra algorithm to calculate the shortest path from the starting point to the goal, with the results displayed in the compiled image. Additionally, the map's resolution was adjusted to align with real-world dimensions, enabling us to modify point values accurately.

2.2. First map and trajectory

We have updated an existing version of the code by creating the trajectory through the midpoints of the common edge between two consecutive polytopes generated from the Dijkstra algorithm. When we first implementede the polytopes, they were square shaped. We used the *CubicSpline* to create the smooth trajectory:

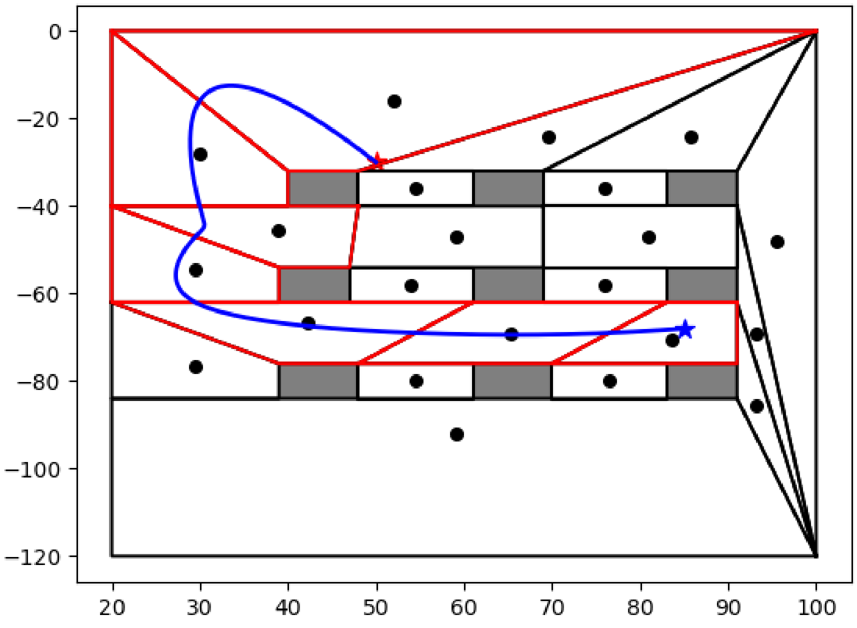


Figure 3.1.: First implementation in TurtleBot world

2.3. Developing the code

We developed the code so that we would not miss too much space near the obstacles and hence we created hexagons around the objects and we used OpenCV, as explained previously, to get the real map. The polytopes were created by getting the middle points of the circles and draw the hexagons by the contour of the objects.

As explained previously, the Dijkstra algorithm would create the fastest path between the 2 points and the path would be highlighted in red.

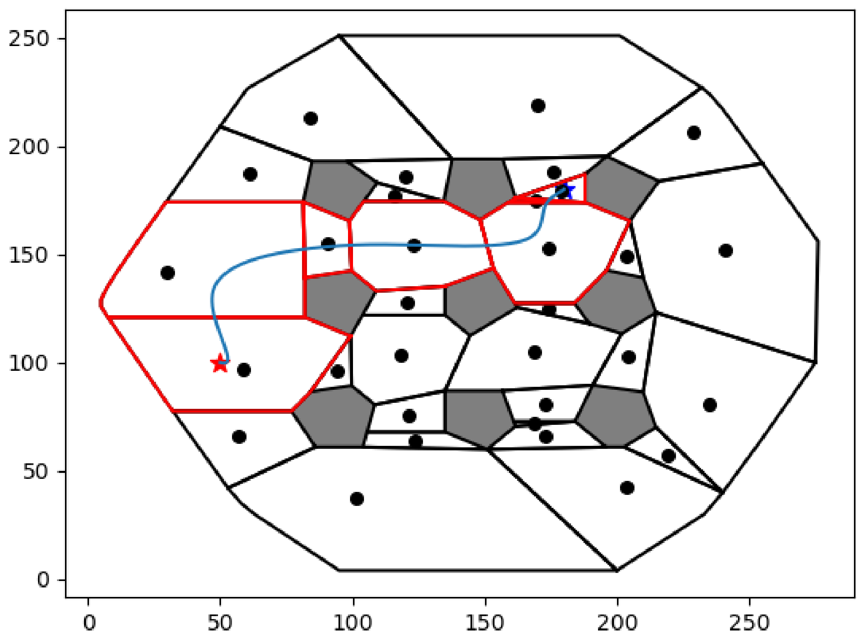


Figure 3.2.: Computing the Trajectory

2.4. ROS2 nodes

We have implemented our own node in which we call the MPC function that follows the trajectory generated by the pre-created function. This code was developed so that we can update the position with the current goal point and give it another goal point to reach. It would have to compute the algorithm in real-time and create the trajectory at each reached goal point.

2.5. Gazebo test

The first test was implemented in the TurtleBot world where we gave it the trajectory created in Fig: 3.2 and we were plotting, as the robot was moving, the states following the reference, but the computation time was affecting the decisions.

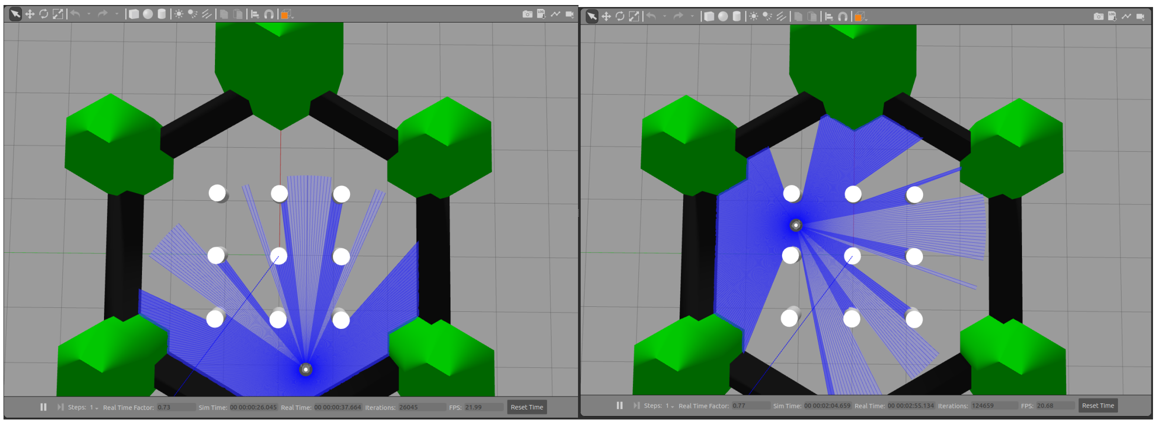

(a) Initial point (b) Goal point

Figure 3.3.: Trajectory tracking

4. Conclusion

1. Future Directions

We used the TurtleBot to create a real-world map; however, limitations in the LiDAR sensor precision and time constraints prevented us from successfully mapping a single room. Our attempts to do so consistently resulted in errors while generating the room's contours and updating the current positions. Additionally, we encountered issues with the odometry queue, which further hindered our ability to compute the map within the available time left.

Moreover, we would want to create the trajectory using the flatness algorithm, so that we could get a better precision in obstacle avoidance.

2. Conclusion

The implementation of an MPC (Model Predictive Control) algorithm with reference tracking for obstacle avoidance on the TurtleBot demonstrated the potential of advanced control strategies in robotic navigation. By integrating reference tracking, the TurtleBot was able to dynamically adjust its trajectory to follow a desired path while effectively avoiding obstacles in its environment. The use of MPC allowed for predictive decision-making, ensuring smooth and efficient navigation by anticipating future states based on the robot's dynamics and sensor data. Despite challenges such as computational demands and real-time tuning, the approach proved robust and adaptable. This work highlights the effectiveness of MPC in achieving precise and safe navigation, laying the groundwork for further enhancements in autonomous robotics applications.

Appendices

A. MATLAB code

```matlab
clc; close all; clear;

% Load CasADi
import casadi.*;

%% Parameters
N_pred_val = 25;       % Prediction horizon value
Q_val = 100;          % State weight value
R_val = 1;             % Input weight value
P_val = 1000;          % Terminal state value

% Define prediction and simulation steps
Te = 0.1;              % Sampling time
Npred = N_pred_val;    % Prediction horizon
Nsim = 300;            % Number of simulation steps

% Physical parameters
r = 33e-3;            % Radius of wheels
L = 160e-3;           % Distance between wheels

% Define system dimensions
dx = 3;               % State dimensions: x, y, theta
du = 2;               % Control dimensions: V, omega

% Initial condition
x0 = [0.3; 0.2; 0];   % Initial state [x; y; theta]
u0 = zeros(du, 1);    % Initial input [V; omega]

% Reference
t = 0 : 2 * pi / (Nsim + Npred) : 2 * pi;
xref = [sin(t); cos(t); zeros(1, length(t))];

% Constraints
Vmin = -10; Vmax = 10;            % Velocity limits
omegamin = -10; omegamax = 10;    % Angular velocity limits
xmin = -1 * 0.2 * ones(3, 1);     % State limits
xmax =  1 * 0.2 * ones(3, 1);

% Weights for cost function
% Q = Q_val * eye(dx);
Q = diag([100, 100, 0.0001]);
R = R_val * eye(du);
P = P_val * Q;

%% CasADi nonlinear optimization
solver = casadi.Opti();

% Define optimization variables
x = solver.variable(dx, Npred + 1);
u = solver.variable(du, Npred);

% Define initial state as a parameter
```

```
xinit = solver.parameter(dx, 1);

% Nonlinear dynamics
f_dynamics = @(x, u) [ u(1) * cos(x(3));
                       u(1) * sin(x(3));
                       u(2) ];

% Add initial state constraint
solver.subject_to(x(:, 1) == xinit);

% Add constraints and dynamics for each prediction step
for k = 1 : Npred
    % State update constraint using discretized nonlinear dynamics
    solver.subject_to(x(:, k+1) == x(:, k) + Te * f_dynamics(x(:, k), u(:, k)));
    solver.subject_to(xmin <= x(:, k + 1) - x(:, k) <= xmax)

    % Control input constraints
    solver.subject_to(Vmin <= u(1, k) <= Vmax);
    solver.subject_to(omegamin <= u(2, k) <= omegamax);
end

%% Define the objective function
% Define the solver
options.ipopt.print_level = 0;
options.ipopt.sb= 'yes';
options.print_time = 0;

% simulation loop
usim = zeros(du, Nsim);
xsim = zeros(dx, Nsim);
xsim(:, 1) = x0;
usim_init = u0;

t1 = tic();
for i = 1 : Nsim
    objective = 0;
    for k = 1 : Npred
        if k ~= 1
            objective = objective + (x(:, k) - xref(:, i + k - 1))' * Q * (x(:, k) - xref(:, i + k - 1)) ...
                + (u(:, k) - u(:, k - 1))' * ...
                R * (u(:, k) - u(:, k - 1));
        else
            objective = objective + (x(:, k) - xref(:, i + k - 1))' * Q * (x(:, k) - xref(:, i + k - 1)) ...
                + u(:, k)' * R * u(:, k);
        end
    end

    objective = objective + (x(:, Npred + 1) - xref(:, i + Npred))' * P * (x(:, Npred + 1) ...
        - xref(:, i + Npred));
    solver.minimize(objective)
    t2 = toc(t1);
    solver.solver('ipopt', options)
    solver.set_value(xinit, xsim(:, i))
    sol = solver.solve();
    usol = sol.value(u);
    usim(:, i) = usol(:, 1);
    xsim(:, i + 1) = xsim(:, i) + Te * f_dynamics(xsim(:, i), usim(:, i)
   );

end

%% Plot results
figure
stem(xsim(1, :));
hold on
stem(xsim(2, :));
hold on
plot(xref(1, 1:Nsim))
hold on
plot(xref(2, 1:Nsim))
grid
```

B. Reference generating (taken from previous student)

```python
import cv2
import numpy as np

import matplotlib.pyplot as plt
import gdspy
from poly_decomp import poly_decomp as pd
from matplotlib.patches import Circle, Wedge, Polygon
from numpy import linalg as LA
#import win32clipboard as clipboard
from scipy.sparse.csgraph import shortest_path
from scipy.interpolate import BSpline

def compare_2_areas_get_1(contour_1, contour_2):
    area_1 = cv2.contourArea(contour_1)
    area_2 = cv2.contourArea(contour_2)
    max_ = max(area_1, area_2)
    return max_


def plot_poly_map(ws, ax, col):
    # plot the whole polytope map in subplot ax with color col
    for i in range(len(ws)):
        x_de = [ws[i][k][0] for k in range(len(ws[i]))]
        x_de.append(ws[i][0][0])
        y_de = [ws[i][k][1] for k in range(len(ws[i]))]
        y_de.append(ws[i][0][1])
        ax.plot(x_de, y_de, color=col)
    return 0

# Check if two decomposed polytopes lies next to each other
# if the there are exactly 2 vertices of one inside the other ===> true
def check_consecutive_polytopes(p1, p2):
    check = False
    s = 0
    check_list = gdspy.inside(p1, gdspy.Polygon(p2))
    for i in range(len(check_list)):
        if check_list[i]:
            s = s + 1
    if s == 2:
        check = True
    return check


def center(polyp):
    xc = np.matrix(polyp)
    return xc.mean(0)


def find_polyp(workspace, points):  ######### check point in polygon
```

```python
    polyp_index = -1
    for i in range(len(workspace)):
        if (gdspy.inside([points], gdspy.Polygon(ws[i])))[0]:
            polyp_index = i
    return polyp_index


def generate_sequence(workspace, init_pts, goal_pts):
    init_polyp = find_polyp(workspace, init_pts)
    goal_polyp = find_polyp(workspace, goal_pts)
    seq = [init_polyp, goal_polyp]
    return seq

def get_path(Pr, i, j):
    path = [j]
    k = j
    while Pr[i, k] != -9999:
        path.append(Pr[i, k])
        k = Pr[i, k]
    return path[::-1]


print(cv2.__version__)
resolution = 0.1   # resolution in file .yaml
image = cv2.imread('/home/truontmh/python_code/binary_map_after_gimp.png')
# convert image to gray image
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

height, width = image.shape
print(f'Height in real environment: {height * resolution} (m)')
print(f'width in real environment: {width * resolution} (m)')

# set a threshold, convert image to binary image (0 and 255)
# 0 is black, 255 is white
threshold = 127
_,binary_image = cv2.threshold(image, threshold,255,cv2.THRESH_BINARY)
image = binary_image

########### find contour of black pixels  ###########

inverted_image = cv2.bitwise_not(image) # function cv2.findContours will find the contours of white pixels
contours,_ = cv2.findContours(inverted_image, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
print("len of contours: ", len(contours))

########### compute areas of contours ###############
area_list = []
for contour in contours:
    area = cv2.contourArea(contour)
    area_list.append(area)

max_area = max(area_list)
max_area_index = area_list.index(max_area)

area_list[max_area_index] = 0 # convert max value to 0, -> find the second largest value
second_max_area = max(area_list)
second_max_area_index = area_list.index(second_max_area)
```

```python

########### remove largest and second largest contour ############
filter_contours = [x for i,x in enumerate(contours) if i not in (max_area_index, second_max_area_index)]
filter_area = [x for i,x in enumerate(area_list) if i not in (max_area_index, second_max_area_index)]


########### compute moments of contours ##############
centroid_list = []
for contour in filter_contours:
    moment = cv2.moments(contour)
    # Compute centroids from moments
    if moment['m00'] != 0:
        cX = int(moment['m10']/moment['m00'])
        cY = int(moment['m01']/moment['m00'])
    else:
        cX, cY = 0,0  # if the contour doesn't have area
    centroid_list.append([cX,cY])


########### Find contour pairs with close center points. After that, chose one which have larger area.
thresh_distance = 5
get_contour_index_list = []
for i in range(len(centroid_list)):
    for j in range(len(centroid_list)):
        if i != j:
            point_1 = np.array(centroid_list[i])
            point_2 = np.array(centroid_list[j])
            distance = np.linalg.norm(point_2 - point_1)
            if distance < thresh_distance:
                max_ = compare_2_areas_get_1(filter_contours[i], filter_contours[j])
                if max_ == filter_area[i] and i not in get_contour_index_list:
                    get_contour_index_list.append(i)
                elif max_ == filter_area[j] and j not in get_contour_index_list:
                    get_contour_index_list.append(j)



############ find convex for contours
hull = []
hull.append(cv2.convexHull(contours[max_area_index]))
for index in get_contour_index_list:
    hull.append(cv2.convexHull(filter_contours[index]))

############ reduce the number of point of each hull -> reduce the computations
approx_hull_list = []
for convex_hull in hull:
    convex_hull = convex_hull.reshape(-1,1,2)
    desired_vertices = len(convex_hull)//1.67
    epsilon = 0.01 * cv2.arcLength(convex_hull, True)
    approx_hull = convex_hull
    while len(approx_hull) > desired_vertices:
        epsilon += 0.01 * cv2.arcLength(convex_hull, True)
        approx_hull = cv2.approxPolyDP(convex_hull, epsilon, True)
    while len(approx_hull) < desired_vertices:
        epsilon = epsilon/1.01
        approx_hull = cv2.approxPolyDP(convex_hull, epsilon, True)
    approx_hull_list.append(approx_hull)





############ Create polytope
obstacles_list = []
hole_list = []
hole_large_list = []
safety_offset = 5


for i in range(len(approx_hull_list)):
    if i == 0:
        ws_limit = np.array(approx_hull_list[0])
        ws_limit = [tuple(x[0]) for x in ws_limit]
    if i!=0:
        obstacle = approx_hull_list[i]
        obstacle = np.array(obstacle)
        obstacle = [tuple(x[0]) for x in obstacle]
        obstacles_list.append(obstacle)




ws_poly = gdspy.Polygon(ws_limit)
for i in range(len(obstacles_list)):
    hole_list.append(gdspy.Polygon(obstacles_list[i]))

# enlarge obstacle a little bit
for i in range(len(hole_list)):
    hole_large_list.append(gdspy.offset(hole_list[i],safety_offset))

# subtraction
poly_with_hole = gdspy.boolean(ws_poly, hole_large_list, "not")

# decomposition
ws = pd.polygonQuickDecomp(poly_with_hole.polygons[0])

# initial point and goal
init = [100, 50]
goal = [186,182]
n_polyp = len(ws)
init_goal_idx = (generate_sequence(ws, init, goal))
center_list = []
for j in range(n_polyp):
    center_list.append(center(ws[j]).A1)


# print(center_list[0])
N = np.empty((0, n_polyp), float)
#print(f'ahihihihi {N.shape}')
N_temp = np.array([])
for i in range(n_polyp):
    N_temp = []
    for j in range(n_polyp):
        if check_consecutive_polytopes(ws[i], ws[j]) and (i != j):
            temp = [center_list[i], center_list[j]]
            N_temp.append(LA.norm(temp))
        else:
            N_temp.append(0)
    N = np.append(N, [N_temp], axis=0)
#print(N)
# D i j k s t r a s algorithm
D, Pr = shortest_path(N, directed=False, method='D', return_predecessors
    =True)
sequence = get_path(Pr, init_goal_idx[0], init_goal_idx[1])

# create b_spline curve
sequence_point = []
sequence_point.append(init)
for i in sequence:
    sequence_point.append(center_list[i])
sequence_point.append(goal)

points = np.array(sequence_point)
degree = 3
t = np.linspace(0, 1, len(points) + degree + 1 - 2*degree)
knots = np.concatenate(([0]*degree, t, [1]*degree))
bspline = BSpline(knots, points, degree)
t_eval = np.linspace(0, 1, 1000)
spline_points = bspline(t_eval)






fig = plt.figure()
ax = fig.add_subplot(111)
plot_poly_map(ws, ax, 'black')
# plot_poly_map([obstacle1, obstacle2, obstacle3], ax, 'blue')
plt.scatter(init[0], init[1], color='red', s=80, marker='*')
plt.scatter(goal[0], goal[1], color='blue', s=80, marker='*')
plt.scatter([center_list[i][0] for i in range(len(center_list))], [
    center_list[i][1] for i in range(len(center_list))],
            color='k', s=30)
plot_poly_map([ws[i] for i in sequence], ax, 'red')
for hole_large in hole_large_list:
    p = Polygon(hole_large.polygons[0], facecolor = 'k', alpha = 0.5)
    plt.gca().add_patch(p)
# plt.plot(x_fine, y_fine, label='Cubic Spline Trajectory', color='blue
    ')  # trajectory
plt.plot(spline_points[:, 0], spline_points[:, 1], label='B-spline
    trajectory')

plt.show()




# Create a white image
white_image = image.copy()
for i in range(height):
    for j in range(width):
        white_image[i][j] = 255
```

```python
white_image = cv2.cvtColor(white_image, cv2.COLOR_GRAY2BGR)

white_image_2 = white_image.copy()

# Show
cv2.drawContours(white_image, hull, -1, (255,0,0), thickness=2)
cv2.drawContours(white_image_2,approx_hull_list,-1,(0,0,255), thickness
    =2)
cv2.imshow('Image',white_image)
#cv2.imshow('original_image', image)
cv2.imshow('reduce_convex', white_image_2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```