UAV PATH PLANNING WITH OBSTACLE AVOIDANCE

Ahmed Magd Aly Shehata

INNOPOLIS UNIVERSITY CI Project

UAV Path Planning

The problem of the path planning has always grabbed the attention of researchers around the world. In this report I will demonstrate how I approached this problem.

In order to solve the path planning problem, we need to have a starting point, desired point and some representation of the environment that surrounds the UAV (unmanned aerial vehicle). In my project, I have generated a random environment to simulate this process as well be explained in the following sections

1. Generating Random Environment for Testing Purposes:

I have created an algorithm to generate a random environment, and it includes:

a. Bounding region
b. Start point
c. Goal point
d. Obstacles

1.a. Bounding region:

I am generating random rectangular region that UAV should not exceed. It is may be easier to think of it as a bounding room. The following snipped of code shows how I implemented this in MATLAB And I visualized the polytope that is considered as the bounding box using BENSOLVE tools library.

```
%% Generate random environment
%% 1. Random bounding box
rpx = rand*100; % center of bounding box in x
rpy = rand*100; % center of bounding box in y
rpz = rand*100; % center of bounding box in z

% creating bounding box randomly
margin = 15; % margin put to ensure no obstacles on the start and goal points
xlimits = [rpx-50*rand-50 rpx+50*rand+50];
ylimits = [rpy-50*rand-50 rpy+50*rand+50];
zlimits = [rpz-50*rand-50 rpz+50*rand+50];
[bounding_box_x, bounding_box_y, bounding_box_z] = meshgrid(xlimits, ylimits,
zlimits);
```

1.b. Generating starting and goal points:

I have assigned the starting and goal points near the corners of the bounding box. Each one of them is close to corners opposite to each other to ensure that the path that the UAV

follows to go from start to finish passes by all obstacles inside. And this was ensured by the following two lines:

```
%% Start and Goal Points:
start = [xlimits(1)+0.5*rand*(margin), ylimits(1)+0.5*rand*(margin),
zlimits(1)+0.5*rand*(margin)];
goal = [xlimits(2)-0.5*rand*(margin), ylimits(2)-0.5*rand*(margin), zlimits(2)-
0.5*rand*(margin)];
```

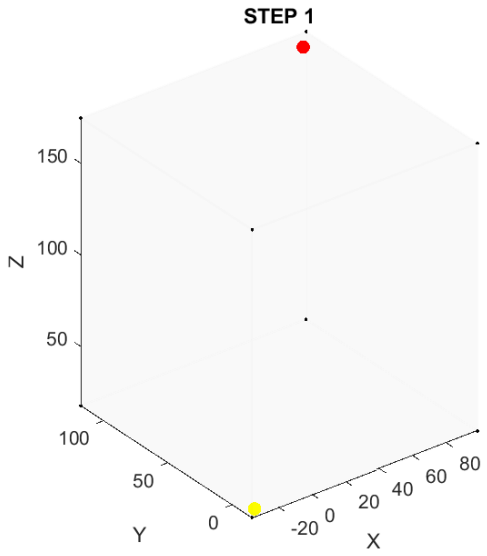


Figure 1 First step in Generating random environment. The gray box is the bounding box, while the yellow and red points are the starting and goal point respecitvely

1.c. Generating obstacles:

I have tried to randomize this process while knowing where are the places that could contain obstacles (this will be helpful for me later on). I have divided the bounding box into a number of grids. This number of grids is randomized as well. Each grid is rectangular box, and the whole bounding box contains $(nGrids)^3$ of such boxes. The obstacles are allowed to be generated inside these grids, and no obstacle can coexists in two grid boxes. And they cannot exist on the very edge of each grid as well. These edges are saved to construct other polytopes that will be discussed later on. The following code shows how these obstacles were generated:

```
%% 2. Random obstacles
xlimits_obstacles_region = [xlimits(1)+margin xlimits(2)-margin];
ylimits_obstacles_region = [ylimits(1)+margin ylimits(2)-margin];
zlimits_obstacles_region = [zlimits(1)+margin zlimits(2)-margin];

nGrids = 3;
xsteps = (xlimits_obstacles_region(2)-xlimits_obstacles_region(1))/nGrids;
ysteps = (ylimits_obstacles_region(2)-ylimits_obstacles_region(1))/nGrids;
zsteps = (zlimits_obstacles_region(2)-zlimits_obstacles_region(1))/nGrids;
```

```
nObstacles = randi(min(20, nGrids^3)); % limit the number of obstacles to 10
obstacles
GridIndices = randi([1, nGrids], 3, nObstacles);
Obstacles_As = {};
Obstacles_bs = {};
Obstacles=[];
nPoints = randi([20, 30]); % number of points to construct an obstacle
for i=1:nObstacles
    Obstacle_ix = (xlimits_obstacles_region(1)+0.5*xsteps +
xsteps*(GridIndices(1,i)-1)) + 0.495*(2*rand(1, nPoints)-1)*xsteps;
    Obstacle_iy = (ylimits_obstacles_region(1)+0.5*ysteps +
ysteps*(GridIndices(2,i)-1)) + 0.495*(2*rand(1, nPoints)-1)*ysteps;
    Obstacle_iz = (zlimits_obstacles_region(1)+0.5*zsteps +
zsteps*(GridIndices(3,i)-1)) + 0.495*(2*rand(1, nPoints)-1)*zsteps;
    Obstacles(:,:,end+1) = [Obstacle_ix; Obstacle_iy; Obstacle_iz];
    rep.V = Obstacles(:,:,end);
    plot(polyh(rep, 'v'), red); hold on
end
```

And every obstacle is generated using random number of points (denoted by nPoints variable). Each obstacle is then appended inside a cell variable so that it could be used later on.

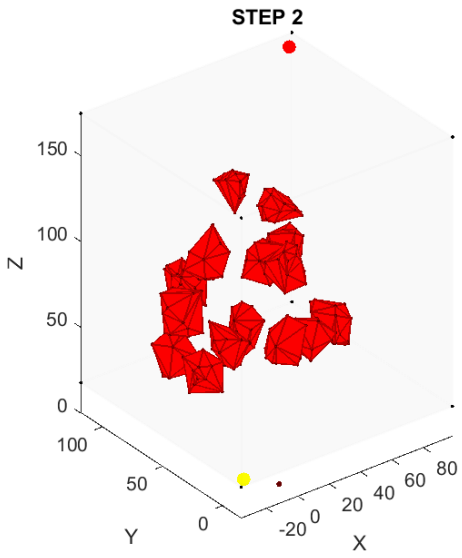


Figure 2 Obstacles generated randomly inside the bounding box

2. Generating collision-free polytopes:

These obstacles cannot be added into the optimization problem directly. This is because if they were added as a constraint, this will take the following form ($Ax \geq b$). And this constraint is non-convex, which implies that it cannot be added and solved using the optimization solvers.

Therefore, this problem could be solved by constructing other polytopes in the free of collision regions and solving the problem as a MICP (Mixid Integer Convex Programming) problem. In this project, I have used a library named IRIS (Iterative Regional Inflation by Semidefinite programming), created by MIT. This library has functions that can help produce polytopes that are free of obstacles and bounded by a bigger polytope (bounding region). This algorithm mainly tries to fit the maximum ellipsoid that does not intersect the obstacles and does not pass the out of the bounding box. Hence, it is required to give a point to that algorithm were it could start iterating to find that maximum ellipsoid that fits that particular region without passing through any obstacles. This point is called the “seed” point. And after that a polytope is constructed using this particular ellipsoid. In my algorithm I have saved the edges of the grid regions, discussed before, for those seeds. This ensures that the seeds are not inside any obstacle.

I have generated two special seeds, one being on top of the starting point and the other is on top of the goal point, because it is mandatory to have obstacle free regions in these regions. The other seeds are planted at the edges of random grids. Also those seeds are planted to be equivalent in number to the obstacles available in the environment. The following snippet shows how I utilized IRIS in my code:

```
%% Generate collision-free polytopes:
[A_bounds, b_bounds] = vert2con(bounding_box.V');
nSeeds = nObstacles; % limit the number of obstacles to 10 obstacles
GridIndices = randi([0, nGrids-1], 3, nSeeds);
As = {};
bs = {};
seeds = [];
seeds_vec = [0.5 0.5 0];
safe_regions = struct('A', {}, 'b', {}, 'C', {}, 'd', {}, 'point', {});
for i=1:nSeeds
    randomized = seeds_vec(randperm(3));
    seed_ix =
(xlimits_obstacles_region(1)+randomized(1)*xsteps+xsteps*(GridIndices(1,i)));
    seed_iy =
(ylimits_obstacles_region(1)+randomized(2)*ysteps+ysteps*(GridIndices(2,i)));
    seed_iz =
(zlimits_obstacles_region(1)+randomized(3)*zsteps+zsteps*(GridIndices(3,i)));
    seeds(end+1,:) = [seed_ix, seed_iy, seed_iz];
    [A, b, C, d, ~] = iris.inflate_region(Obstacles, A_bounds, b_bounds,
seeds(end,:)');
    safe_regions(end+1) = struct('A', A, 'b', b, 'C', C, 'd', d, 'point',
seeds(end,:));
    polytope.B = A; polytope.b = b;
    plot(polyh(polytope, 'h'), green); hold on
    As{end+1}=A;
    bs{end+1}=b;
end
```

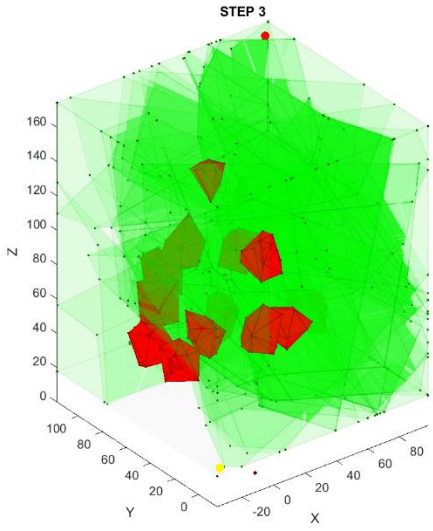


Figure 3 The green areas represent the obstacle free polytopes

3. Path planning:

As discussed before, this problem could be solved using MICP. Therefore, I have used the following optimization problem to find that path:

$$\min_{x1,\dots,x_k} \|x1 - x_{start}\| * w_1 + \left\|x_K - x_{goal}\right\| * w_1 + \sum_{k=1}^{K-1} \|x_{i+1} - x_i\|$$

$$subject\ to \begin{cases} A_{bound} x \le b_{bound} \\ A_i x \le b_i + M * (1 - c_{i,k}), \quad i = 1, \dots, N \\ \sum_{i=1}^{N} c_{i,k} = 1 \\ c_{i,k} \in \{0, 1\} \end{cases}$$

Where c and x are decision variables. C is a matrix of binary variables, the rows represent how many obstacles are there. On the other hand the columns are for the number of steps to be constructed between the starting point and the goal point. The big-M was used here to enforce the points x to be at least inside one of the collision-free polytopes.

This problem was solved using CVX in MATLAB as show below:

```
%% Path planning (Solving the optimization problem):
number_of_steps = 15;
weight_goal = 1000;
weigth_inbetween = 0.1;
bigM = 100;
nPolytopes = length(bs);

cvx_solver mosek
cvx_begin
    variable x(3, number_of_steps)
    binary variable c(nPolytopes, number_of_steps);

    cost = 0;
    for i = 1:(number_of_steps-1)
        cost = cost + pow_pos(norm(x(:, i) - x(:, i+1), 1),
2)*weigth_inbetween;
    end
    cost = cost + pow_pos(norm(x(:, 1) - start'), 2)*weight_goal;
    cost = cost + pow_pos(norm(x(:, number_of_steps) - goal'),
2)*weight_goal;
    start_mat = diag(start);
    goal_mat = diag(goal);
    minimize( cost )
    subject to
        for i = 1:number_of_steps
            A_bounds*x(:, i) <= b_bounds;
            cs=0;
            for j = 1:nPolytopes
                cell2mat(As(j))*x(:, i) <= cell2mat(bs(j)) + (1-
c(j, i))*bigM;
                cs = cs + c(j, i);
            end
            cs == 1;
        end

cvx_end
```

The weights and the M values were tuned to get a good performance in my problem. The first term in the cost function increases whenever the first point gets far away from the starting point. Thus, by multiplying this by w_1, this ensures that the first path point should be as close as possible to the starting point. The same applies for the last point and goal point that are represented in the second norm in the cost function. However, those two terms enough to solve the problem. In order to find a short path that does not collide, we have added the last term. This term ensures that the distance between one point and the next one is being minimized.

On the other hand, the path points should not intersect the obstacles. Which is the same as saying, the path points, should pass only inside the obstacle-free regions, which are represented

by the green polytopes shown in fig. 3. That is why the second constraint is added. And the following figure shows the path that was generated by this optimization problem:

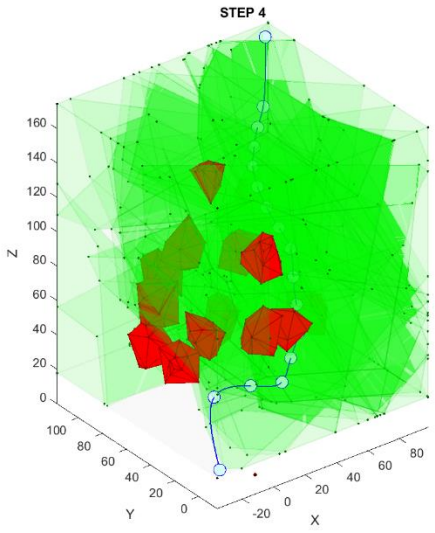


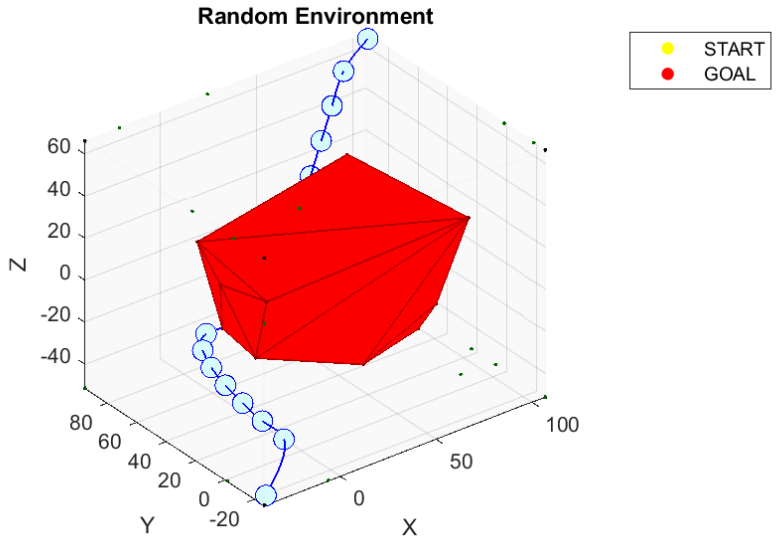
Figure 4 The blue line shows spline fitted to the white points which were obtained by solving the MICP problem

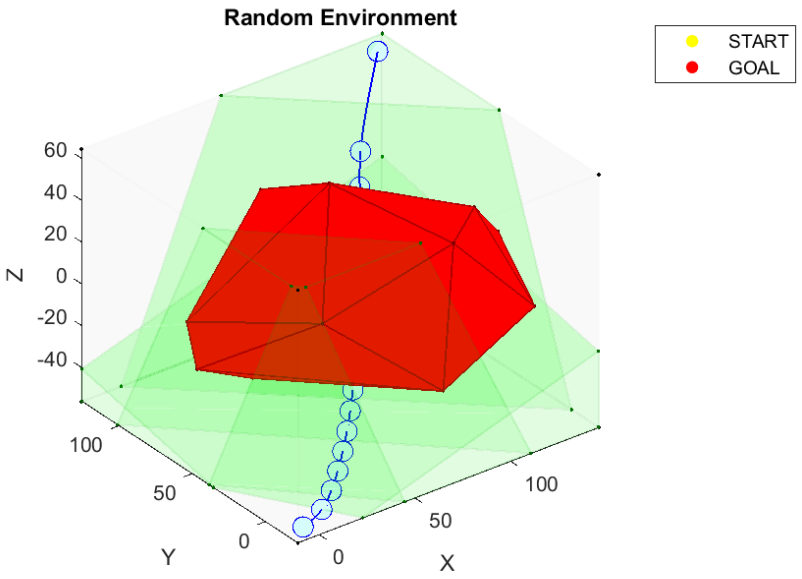
In addition, a spline was fitted through the path points to ensure that the UAV could perform this path. This because according the results published by Mellinger and Kumar, it is not necessary to explicitly consider the dynamics of the drone as long as the path is smooth. Smooth paths could be performed using the drones.

4. Additional Random Tests:

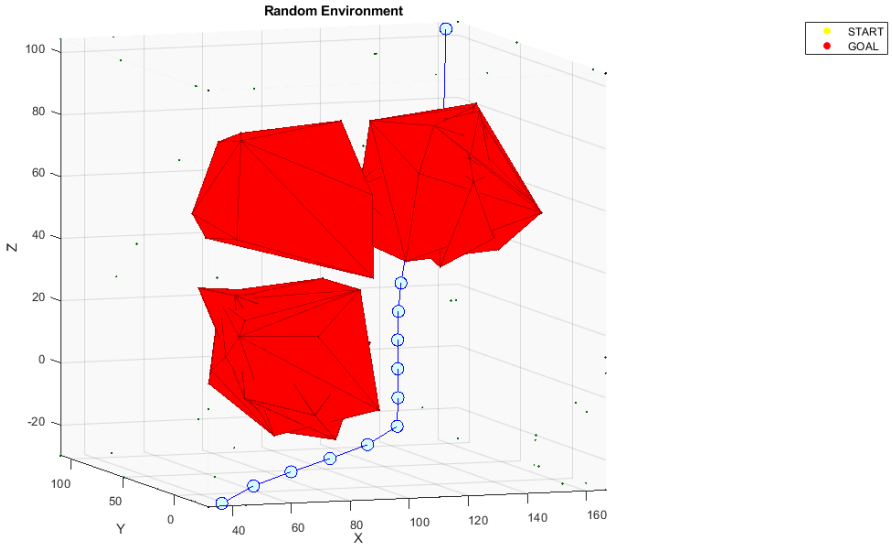
PLEASE IGNORE THE GREEN POINTS IN THE PLOTS, THEY ARE SOME NOISES IN THE PLOTS

- One Obstacle:

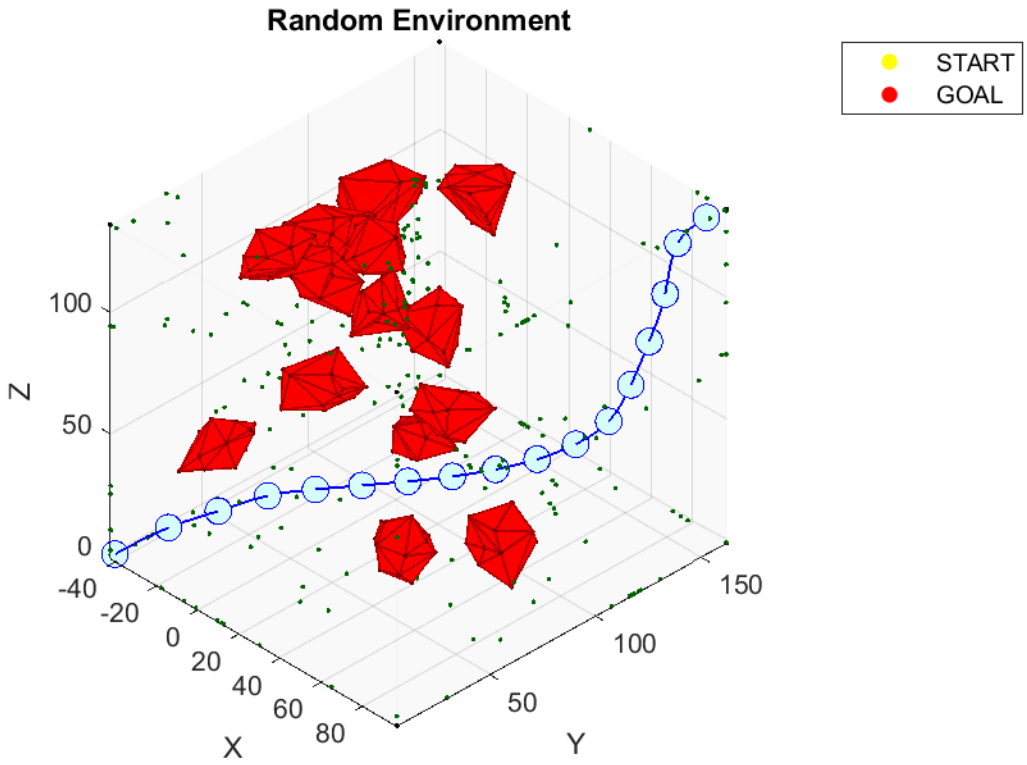


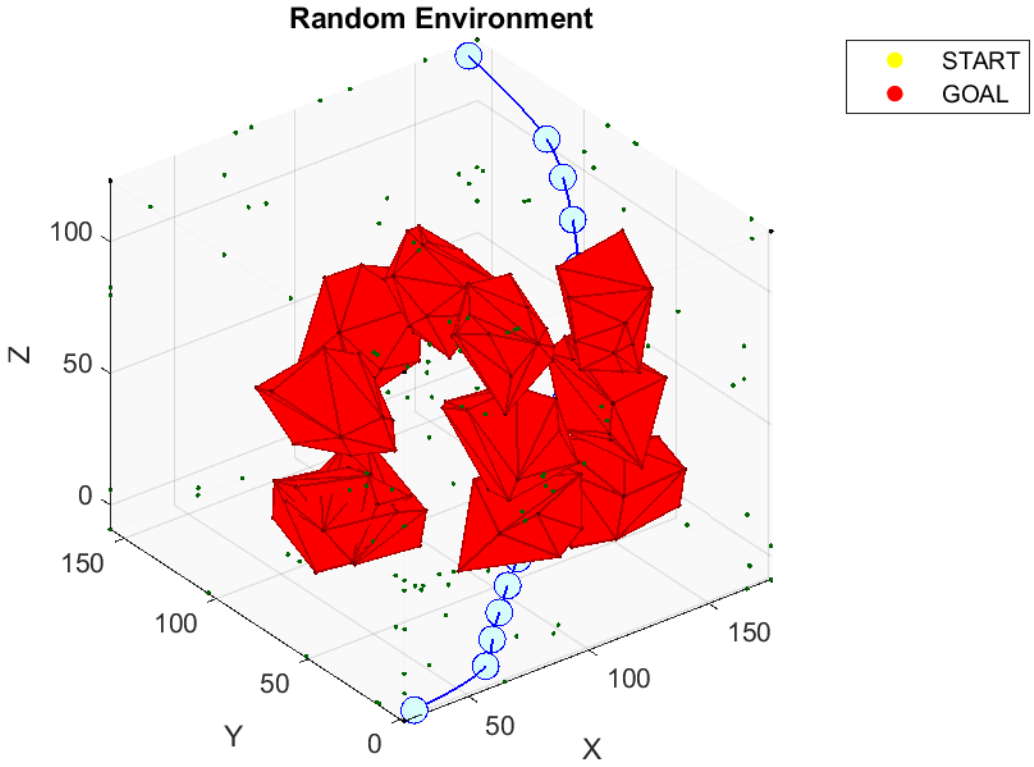


- Three obstacles:



- Many obstacles:


Random Environment
START
GOAL
Z
100
50
0
-40
-20
0
20
40
60
80
X
50
100
150
Y


Random Environment
START
GOAL
Z
100
50
0
150
100
50
0
Y
50
100
150
X

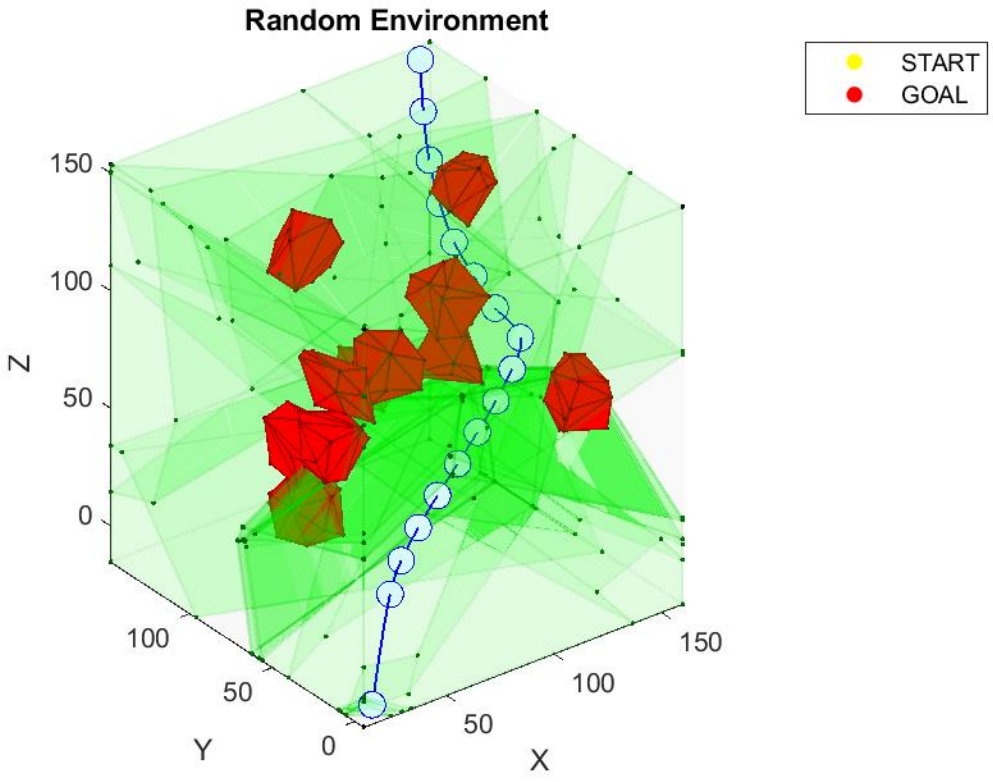

Random Environment
START
GOAL
150
100
50
0
Z
100
50
0
Y
50
100
150
X

Table of Contents


```
clear all
close all
clc
% run('bt-1.3/make_oct.m')
```

Options for plotting polytopes:

```
red.color = [1 0 0];
red.alpha = 1;

gray.color = [0.05 0.05 0.05];
gray.alpha = 0.01;

green.color = [0 1 0];
green.alpha = 0.1;

yellow.color = [1 1 0];
yellow.alpha = 0.2;
```

Generate random environment

1. Random bounding box

```
rpx = rand*100; % center of bounding box in x
rpy = rand*100; % center of bounding box in y
rpz = rand*100; % center of bounding box in z

% creating bounding box randomly
margin = 15; % margin put to ensure no obstacles on the start and goal
 points
xlimits = [rpx-50*rand-50 rpx+50*rand+50];
ylimits = [rpy-50*rand-50 rpy+50*rand+50];
zlimits = [rpz-50*rand-50 rpz+50*rand+50];
[bounding_box_x, bounding_box_y, bounding_box_z] = meshgrid(xlimits,
 ylimits, zlimits);

fake_margin=0.1; % margin used to plot the bounding box such that it
 is not on top of some polytope (asthetics reasons only)
```

```
xlimits_fake = [xlimits(1)-fake_margin xlimits(2)+fake_margin];
ylimits_fake = [ylimits(1)-fake_margin ylimits(2)+fake_margin];
zlimits_fake = [zlimits(1)-fake_margin zlimits(2)+fake_margin];
[bounding_box_x_fake, bounding_box_y_fake, bounding_box_z_fake] =
 meshgrid(xlimits_fake, ylimits_fake, zlimits_fake);

bounding_box.V = [reshape(bounding_box_x,1,[]);
 reshape(bounding_box_y,1,[]); reshape(bounding_box_z,1,[])];
bounding_box_fake.V = [reshape(bounding_box_x_fake,1,[]);
 reshape(bounding_box_y_fake,1,[]); reshape(bounding_box_z_fake,1,
[])];
```

Start and Goal Points:

```
start = [xlimits(1)+0.5*rand*(margin), ylimits(1)+0.5*rand*(margin),
 zlimits(1)+0.5*rand*(margin)];
goal = [xlimits(2)-0.5*rand*(margin), ylimits(2)-0.5*rand*(margin),
 zlimits(2)-0.5*rand*(margin)];
scatter3(start(1), start(2), start(3), 50, 'y', 'filled'); hold on
scatter3(goal(1), goal(2), goal(3), 50, 'r', 'filled'); hold on

plot(polyh(bounding_box_fake,'v'), gray); hold on
```

Undefined function 'polyh' for input arguments of type 'struct'.

Error in Project (line 46)
plot(polyh(bounding_box_fake,'v'), gray); hold on

2. Random obstacles

```
xlimits_obstacles_region = [xlimits(1)+margin xlimits(2)-margin];
ylimits_obstacles_region = [ylimits(1)+margin ylimits(2)-margin];
zlimits_obstacles_region = [zlimits(1)+margin zlimits(2)-margin];

nGrids = randi([1, 5]); % number of grids per axis
xsteps = (xlimits_obstacles_region(2)-xlimits_obstacles_region(1))/
nGrids;
ysteps = (ylimits_obstacles_region(2)-ylimits_obstacles_region(1))/
nGrids;
zsteps = (zlimits_obstacles_region(2)-zlimits_obstacles_region(1))/
nGrids;

nObstacles = randi(min(20, nGrids^3)); % limit the number of obstacles
 to 10 obstacles
GridIndices = randi([1, nGrids], 3, nObstacles);
Obstacles_As = {};
Obstacles_bs = {};
Obstacles=[];
nPoints = randi([20, 30]); % number of points to construct an obstacle
for i=1:nObstacles
    Obstacle_ix = (xlimits_obstacles_region(1)+0.5*xsteps +
 xsteps*(GridIndices(1,i)-1)) + 0.495*(2*rand(1, nPoints)-1)*xsteps;
```

```
    Obstacle_iy = (ylimits_obstacles_region(1)+0.5*ysteps +
 ysteps*(GridIndices(2,i)-1)) + 0.495*(2*rand(1, nPoints)-1)*ysteps;
    Obstacle_iz = (zlimits_obstacles_region(1)+0.5*zsteps +
 zsteps*(GridIndices(3,i)-1)) + 0.495*(2*rand(1, nPoints)-1)*zsteps;
    Obstacles(:,:,end+1) = [Obstacle_ix; Obstacle_iy; Obstacle_iz];
    rep.V = Obstacles(:,:,end);
    plot(polyh(rep, 'v'), red); hold on
end
```

Generate collision-free polytopes:

```
[A_bounds, b_bounds] = vert2con(bounding_box.V');
nSeeds = nObstacles; % limit the number of obstacles to 10 obstacles
GridIndices = randi([0, nGrids-1], 3, nSeeds);
As = {};
bs = {};
seeds = [];
seeds_vec = [0.5 0.5 0];
safe_regions = struct('A', {}, 'b', {}, 'C', {}, 'd', {}, 'point',
 {});
for i=1:nSeeds
    randomized = seeds_vec(randperm(3));
    seed_ix = (xlimits_obstacles_region(1)+randomized(1)*xsteps
+xsteps*(GridIndices(1,i)));
    seed_iy = (ylimits_obstacles_region(1)+randomized(2)*ysteps
+ysteps*(GridIndices(2,i)));
    seed_iz = (zlimits_obstacles_region(1)+randomized(3)*zsteps
+zsteps*(GridIndices(3,i)));
    seeds(end+1,:) = [seed_ix, seed_iy, seed_iz];
    [A, b, C, d, ~] = iris.inflate_region(Obstacles, A_bounds,
 b_bounds, seeds(end,:)');
    safe_regions(end+1) = struct('A', A, 'b', b, 'C', C, 'd',
 d, 'point', seeds(end,:));
    polytope.B = A; polytope.b = b;
    plot(polyh(polytope, 'h'), green); hold on
    As{end+1}=A;
    bs{end+1}=b;
end

[A_start, b_start, C, d, ~] = iris.inflate_region(Obstacles, A_bounds,
 b_bounds, start');
safe_regions(end+1) = struct('A', A_start, 'b', b_start, 'C', C, 'd',
 d, 'point', start);
polytope.B = A_start; polytope.b = b_start;
plot(polyh(polytope, 'h'), green); hold on
As{end+1}=A_start;
bs{end+1}=b_start;

[A_goal, b_goal, C, d, ~] = iris.inflate_region(Obstacles, A_bounds,
 b_bounds, goal');
safe_regions(end+1) = struct('A', A_goal, 'b', b_goal, 'C', C, 'd',
 d, 'point', goal);
polytope.B = A_goal; polytope.b = b_goal;
```

```
plot(polyh(polytope, 'h'), green); hold on
As{end+1}=A_goal;
bs{end+1}=b_goal;
```

Path planning (Solving the optimization problem):

```
number_of_steps = 15;
weight_goal = 1000;
weigth_inbetween = 0.1;
bigM = 100;
nPolytopes = length(bs);

cvx_solver mosek
cvx_begin
    variable x(3, number_of_steps)
    binary variable c(nPolytopes, number_of_steps);

    cost = 0;
    for i = 1:(number_of_steps-1)
        cost = cost + pow_pos(norm(x(:, i) - x(:, i+1), 1),
 2)*weigth_inbetween;
    end
    cost = cost + pow_pos(norm(x(:, 1) - start'), 2)*weight_goal;
    cost = cost + pow_pos(norm(x(:, number_of_steps) - goal'),
 2)*weight_goal;
    start_mat = diag(start);
    goal_mat = diag(goal);
    minimize( cost )
    subject to
        for i = 1:number_of_steps
            A_bounds*x(:, i) <= b_bounds;
            cs=0;
            for j = 1:nPolytopes
                cell2mat(As(j))*x(:, i) <= cell2mat(bs(j)) + (1-c(j,
 i))*bigM;
                cs = cs + c(j, i);
            end
            cs == 1;
        end

cvx_end
path = [start', x, goal'];
plot3(path(1, :), path(2, :),
 path(3, :),'o','Color','b','MarkerSize',10,'MarkerFaceColor','#D9FFFF')
fnplt(cscvn(path),'b',1)
```

Plot variables:

```
title('Random Environment');
xlabel('X');
```

```
ylabel('Y');
zlabel('Z');
axis equal
legend('START', 'GOAL')
grid off
```

Published with MATLAB® R2019b