RBE550 – Motion Planning

Assignment: **Wildfire**

Submitted by:
Suketu Parekh

➢ Workflow Overview:

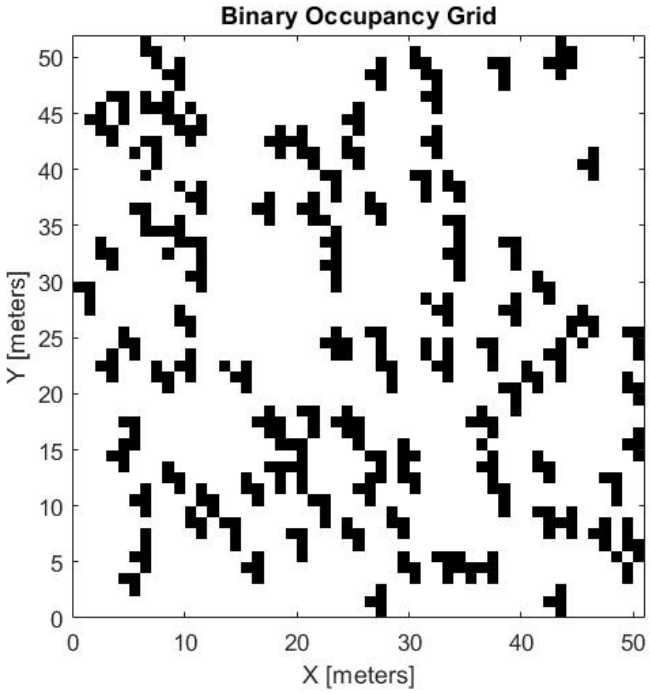
The following steps were considered for the overall fire-fighting operation using different types of path planners for the path planning of the fire truck:

1. Setup the environment to act as a base for the planning process
2. **Inflate** the obstacle portion of the Environment to account for the size of different vehicles and avoid collision – **Minkowski sum**
3. Serve the code to the Sampling-Based planner algorithm to make the roadmap a priori
4. Coding the environment for making it dynamic for the spreading of fire
5. Serving the environment to the planners for the path-planning operation
6. Employ a **Pure Pursuit controller** and run it on the generated path taking into account the Kinematics of vehicle
7. Animating the final results for the paths and time

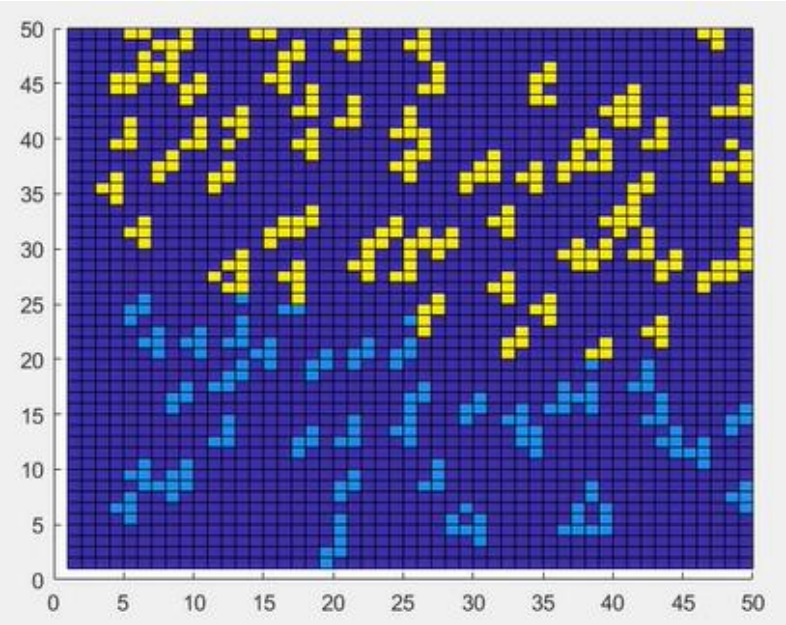
➢ Environment Setup:

The environment was setup in MATLAB by forming a 50-by-50 matrix and randomly placing the tetris shaped obstacles to make 20% occupancy and displaying out its Binary Occupancy Map as shown in the following image.

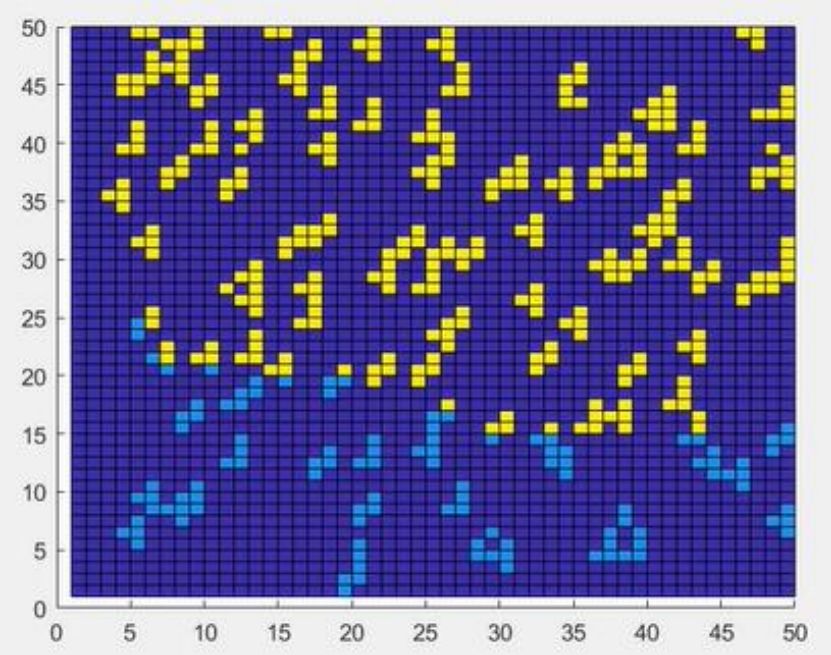
The size of the Forest environment was taken as 50-by-50 as the whole environment was scaled down by a factor of 5. The vehicle will be spawned at coordinates of (2,2).



The environment was coded to spread-out the fire to other obstacles within a radius of 6 cells (6*5=30m) of the initial fire. The same is then repeated at every 20 second interval and the neighboring obstacles of a burning obstacle catch fire as well. The following images display the same. Yellow cells are the cells set to burning state. Blue obstacles are still intact.



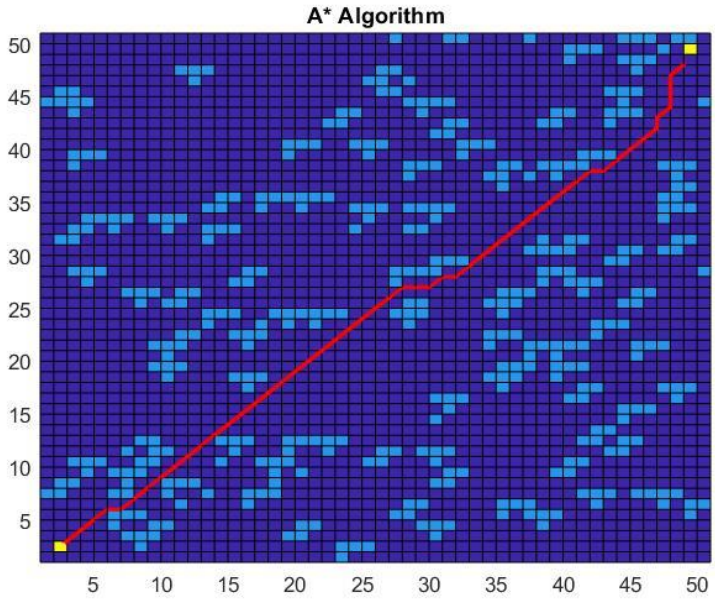
Time = t



Time = t + 20sec

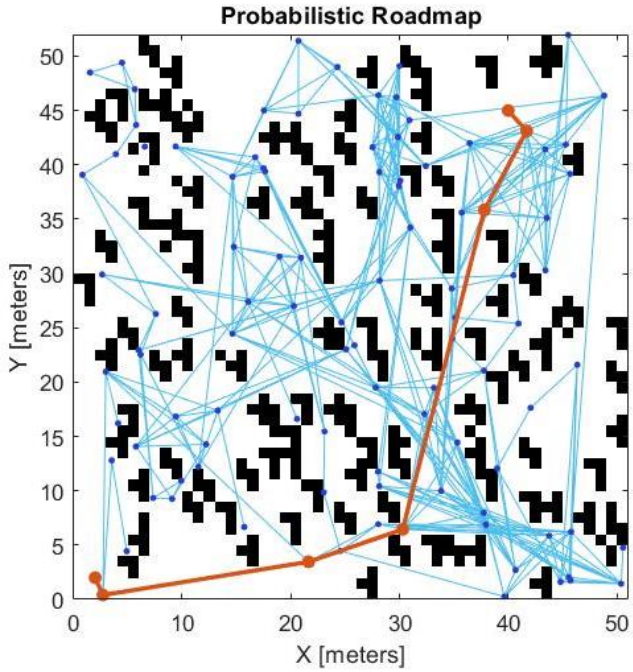
➢ Combinatorial Planer Implementation:

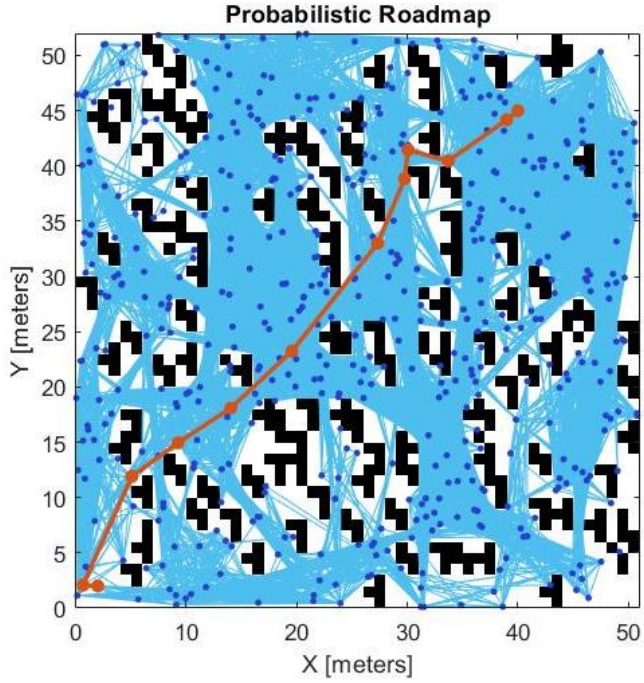
A combinatorial planer implementation was employed for the planner. An A* algorithm was used. The following the output of the A* algorithm implementation on a test case.



➢ Sampling-Based Planner Implementation:

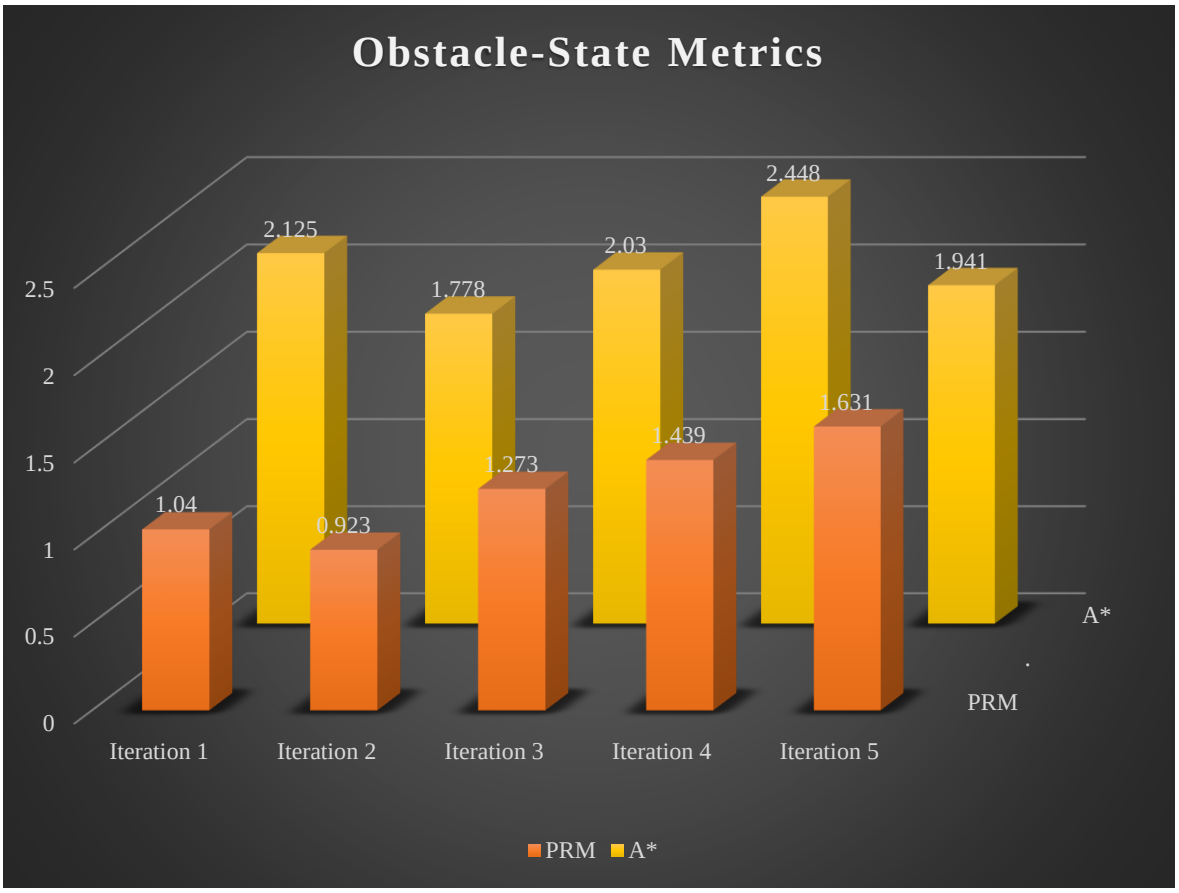
A Probabilistic Roadmap (PRM) was created a priori for the Forest environment. Different number of nodes were experimented with to find a good number of nodes to be considered. The results of the same can be seen in the following figures with 100 and 500 nodes respectively.





➢ Obstacle-State Metrics:

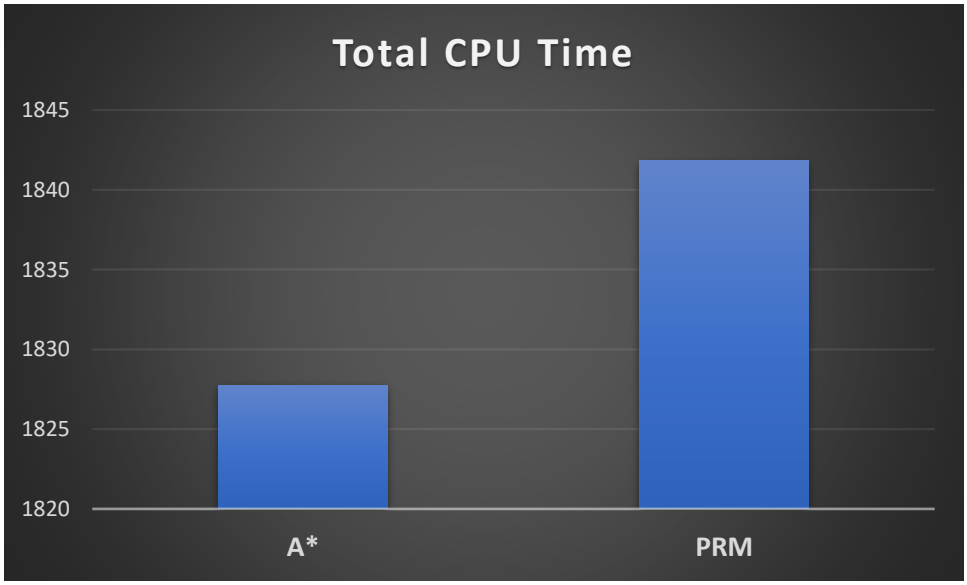
Iteration	Combinatorial Planner		Sampling-Based Planner	
	N_{intact}/N_{total}	$N_{exhausted}/N_{burned}$	N_{intact}/N_{total}	$N_{exhausted}/N_{burned}$
1	0	2.125	0	1.040
2	0	1.778	0	0.923
3	0	2.030	0	1.273
4	0	2.448	0	1.439
5	0	1.941	0	1.631
Total	**0**	**10.332**	**0**	**6.306**



➢ CPU Time:

The total CPU runtime for both the planner types was found using the *tic; toc* functions in MATALB that measure the total execution time between the lines of code placed between the expressions. The following are the mentioned runtimes for the planners when executed at 10x speed that is, ideally simulated for 360 seconds. Also, the times mentioned do not include any sort of plotting of the figures/plots.

Iteration	Combinatorial Planner	Sampling-Based Planner
1	367.342	370.102
2	363.113	367.189
3	368.308	366.983
4	364.008	371.705
5	364.980	365.859
Total	1827.751	1841.838

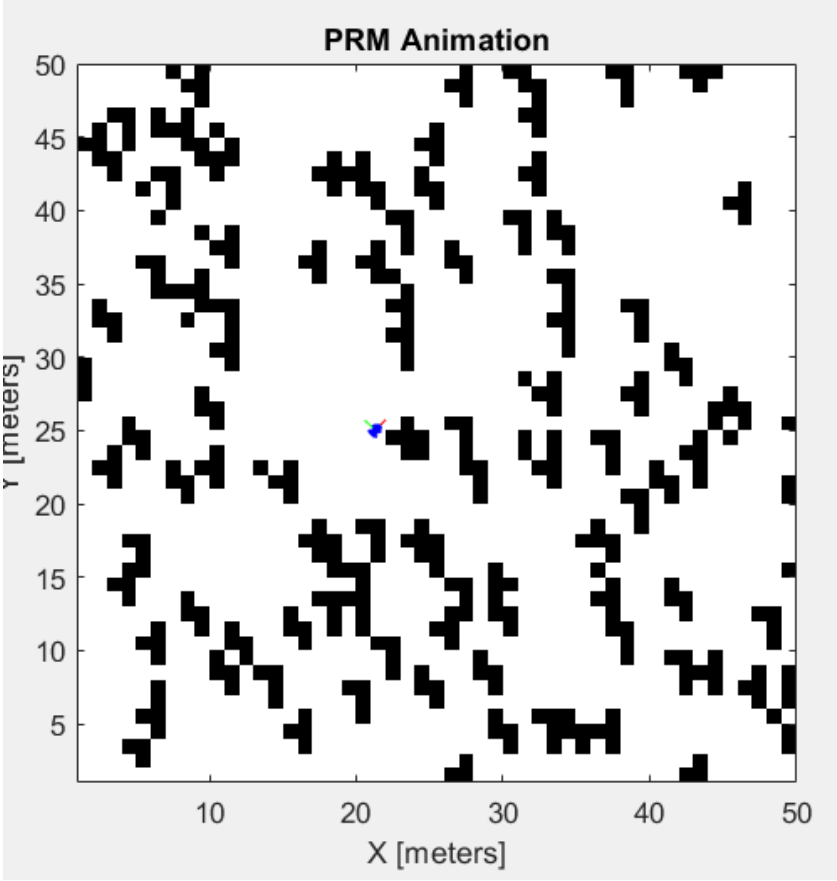


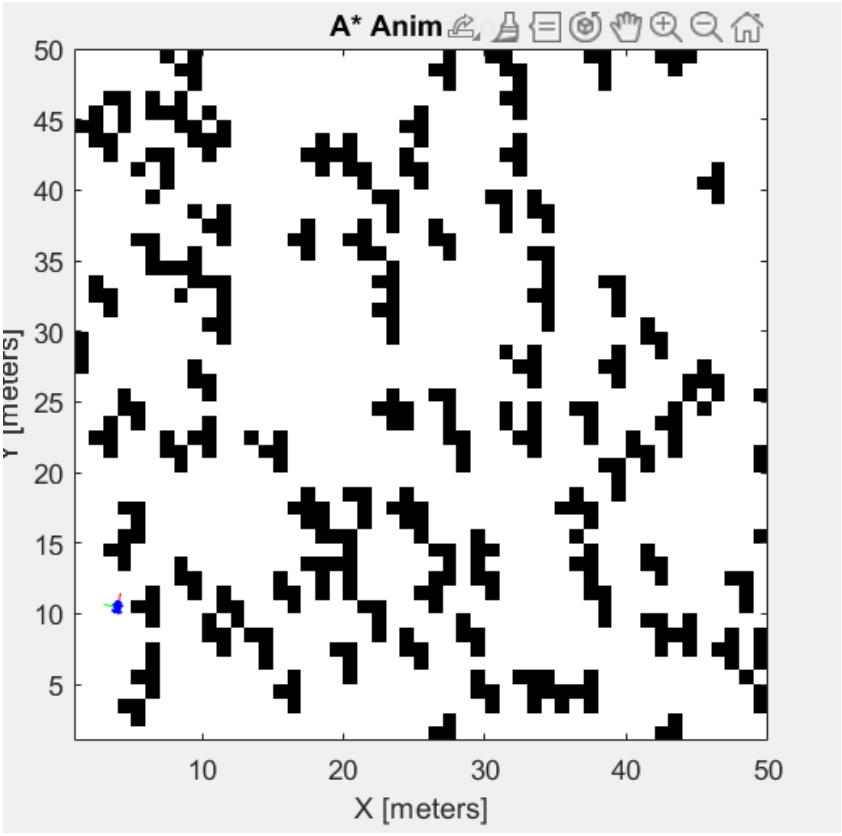
➢ Animation:

The dynamic visualization was carried out for one of the fire-fighting iterations and the fire truck was animated on the Forest environment. Since the animation was run at 10x speed, the maximum linear velocity of the truck was taken as 100m/s and the animation was run with the mentioned events

happening at $1/10^{th}$ of the specified time. This resulted in the required animation at the required speed.

One of the setbacks of the animation is that the state of obstacles (burning, extinguished or intact) at any time can not be known from the animation itself. It was because of the fact that display of *binaryOccupancyMap* variables does not allow any state other than 0 (white) or 1 (black). The *surf* function in MATLAB would allow for showing of the states of obstacles but it won't allow for the vehicle to be plotted on the same. So a trade-off had to be made and the showing of the states of the obstacles was eliminated. The same can be seen in the following images.





➢ Observations and Results:

As seen from the Obstacle State Metrics, the A* algorithm does a pretty good job of extinguishing fire with an average of ~70% of the fire extinguished by the end of the simulation. This can be because of the fact that A* algorithm always guarantees to find the shortest path between 2 points. For the PRM algorithm, the extinguishing was at ~55% on average. This is because of the fact that the algorithm finds a random-possible path which may not be optimal in most of the cases. Also, the PRM algorithm took longer for the test run as compare to the A* algorithm.

```
>> ProbRoadMap
Elapsed time is 3.050708 seconds.
>>
```

```
>> Astar_path
Elapsed time is 0.165361 seconds.
>>
```

Thus, as a clear result of the observations, it can be concluded that the A* algorithm (Combinatorial Planner) is a better choice of planner for the given application as compared to the PRM (Sampling-based Planner) algorithm.

➢ References:

1. Anthony Chrabieh (2021). A star search algorithm (https://www.mathworks.com/matlabcentral/fileexchange/64978-a-star-searchalgorithm), MATLAB Central File Exchange. Retrieved November 15, 2021.

➢ Appendices:

1. Creating Obstacle field: Obstacle.m

```
prompt = 'Enter Grid Length:'; %of occupied cells in the grid
s = input(prompt);

size = s*s;

Tet_A = [1 1 1 1]'; %Verticle Block
occ_A = 1;

Tet_B = [1 1 1 1];  %Horizontal Block
occ_B = 1;

Tet_C = [1 1; 1 1]; %Square Block
occ_C = 1;

Tet_D = [1 1;0 1;0 1]; %eg2
occ_D = 4/6;

Tet_E = [1 0;1 1;0 1]; %eg3
occ_E = 4/6;

Tet_F = [0 1;1 1;0 1]; %eg4
occ_F = 4/6;

Tet_G = [0]; %blank
occ_G = 0;

prompt = 'Enter Occupancy from 0 to 1:'; %of occupied cells in the grid
occ_Eff = input(prompt);

syms x y z
assume(x,'integer')
assumeAlso(x >= 0) %to make the grid a bit more intereting
assume(y,'integer')
if occ_Eff >= 0.1 && occ_Eff <= 0.9
    assumeAlso(y >= 3) %to make the grid a bit more intereting
end
assume(z,'integer')
assumeAlso(z >= 0)

LHS = floor(occ_Eff*size);
diff = mod(LHS,4);
LHS = LHS - diff;

eqn1 = LHS == (x*occ_A*4 + y*occ_D*6 + z*occ_G);
eqn2 = size == 4*x + 6*y + z;
sol = solve([eqn1 eqn2],[x y z]);
x = sol.x;
y = sol.y;
z = sol.z;
```

```
x(1);
y(1);
z(1);

a = floor(x(1)/3);
b = floor(x(1)/3);
c = floor(x(1)/3);
d = floor(y(1)/3);
e = floor(y(1)/3);
f = floor(y(1)/3);
g = z(1);

T = [a 3 0; b 0 3; c 1 1; d 2 1; e 2 1; f 2 1; g 0 0];

OCPD = zeros(s,s);

i = 1; %varies from 1-number of a particular type of block
j = 1; %varies from 1-7 // index of block type

while (i<=T(j,1))
    cord = floor(s*rand(1,2));
    OCPD(cord(1,1)+1:(cord(1,1)+T(j,3)+1), cord(1,2)+1:(cord(1,2)+T(j,2)+1))=Tet_A;
    i=i+1;
end
i = 1;
j=j+1;

while (i<=T(j,1))
    cord = floor(s*rand(1,2));
    OCPD(cord(1,1)+1:(cord(1,1)+T(j,3)+1), cord(1,2)+1:(cord(1,2)+T(j,2)+1))=Tet_B;
    i=i+1;
end
i = 1;
j=j+1;

while (i<=T(j,1))
    cord = floor(s*rand(1,2));
    OCPD(cord(1,1)+1:(cord(1,1)+T(j,3)+1), cord(1,2)+1:(cord(1,2)+T(j,2)+1))=Tet_C;
    i=i+1;
end
i = 1;
j=j+1;

while (i<=T(j,1))
    cord = floor(s*rand(1,2));
    OCPD(cord(1,1)+1:(cord(1,1)+T(j,2)+1), cord(1,2)+1:(cord(1,2)+T(j,3)+1))=Tet_D;
    i=i+1;
end
i = 1;
j=j+1;

while (i<=T(j,1))
    cord = floor(s*rand(1,2));
    OCPD(cord(1,1)+1:(cord(1,1)+T(j,2)+1), cord(1,2)+1:(cord(1,2)+T(j,3)+1))=Tet_E;
    i=i+1;
```

```
end
i = 1;
j=j+1;

while (i<=T(j,1))
    cord = floor(s*rand(1,2));
    OCPD(cord(1,1)+1:(cord(1,1)+T(j,2)+1), cord(1,2)+1:(cord(1,2)+T(j,3)+1))=Tet_F;
    i=i+1;
end

r = s+2;                          % Get the matrix size
c = s+2;

show(binaryOccupancyMap(OCPD))

% prm = mobileRobotPRM(binaryOccupancyMap(OCPD),500)
% show(prm)
```

2. Making Dynamic Fire Spreading in Forest: Forest.m

```
clear list start intrm time frst timeline

frst = OCPD;
timeline(1,:,:) = OCPD;

list(1,1)=0;
list(1,2)=0;

for i=1:50
    for j=1:50
        if frst(i,j)==1
            list(end+1,:)=[i,j];
        end
    end
end

list(1,:) = [];

init = floor(length(list)*rand);
initr = list(init,1);
initc = list(init,2);

% start=tic;
%
% time=toc(start);
%
%     while (time<6)
%         time=toc(start);
%     end
%
%     if (time>6 && time<6.1)
frst(initr,initc)=3; % burning state
%     end
```

```
timeline(2,:,:) = frst;
%     surf(frst)
%     view(2)
%     pause(1)
%     dur=toc(start);

%     while(dur<=20)
%         if(mod(dur,2)<=0.0001)
% spread fire to radius of 30m
n=1;
while(n>0)
    frst = spreadFire(frst,list);
    % mod(dur,2)
    timeline(end+1,:,:)=frst;
    pause(2)
    %             surf(frst)
    %             view(2)
    %             pause(1.75);
    %         end
    %         dur=toc(start);
end
```

3. Function: spreadFire.m

```
function sf = spreadFire(OCPD,list)
n=0;
for i=1:length(list)
    if OCPD(list(i,1),list(i,2))==3
        (list(i,1),list(i,2))
        for j=list(i,1)-5:list(i,1)+5
            for k=list(i,2)-5:list(i,2)+5
                if k<51 & k>0 & j<51 & j>0 & OCPD(j,k)==1
                    OCPD(j,k)=3;
                    n=1;
                end
            end
        end
    end
end

sf = OCPD;
% n = n;
% for i = cntr(1)-n:cntr(1)+n
%     for j = cntr(2)-n:cntr(2)+n
%         if i<51 & i>0 & j<51 & j>0 & OCPD(i,j)==1
%             OCPD(i,j)=3;
%         end
%     end
% end

end
```

4. ProbRoadMap.m

```
% startLocation = [2,2];
% endLocation = [40,45];
```

```
map=binaryOccupancyMap(OCPD);
show(map)

% mapInflated = copy(map);
% inflate(mapInflated,0.01);

prm = mobileRobotPRM(map,500);
path = findpath(prm, startLocation, endLocation);

% show(prm)
% plot(path(:,1),path(:,2))
% xlim([1,50])
% ylim([1,50])
% show(binaryOccupancyMap(OCPD));
% hold on

bicycle =
bicycleKinematics("VehicleInputs","VehicleSpeedHeadingRate","MaxSteeringAngle",pi/8);
waypoints = path;
sampleTime = 0.05;               % Sample time [s]
vizRate = rateControl(30/sampleTime);
tVec = 0:sampleTime:20;          % Time array
initPose = [waypoints(1,:)'; 0]; % Initial pose (x y theta)
controller2 =
controllerPurePursuit("Waypoints",waypoints,"DesiredLinearVelocity",3,"MaxAngularVelo
city",3*pi);
goalPoints = waypoints(end,:)';
goalRadius = 1.5;
[tBicycle,bicyclePose] =
ode45(@(t,y)derivative(bicycle,y,HelperMobileRobotController(controller2,y,goalPoints
,goalRadius)),tVec,initPose);
bicycleTranslations = [bicyclePose(:,1:2) zeros(length(bicyclePose),1)];
bicycleRot = axang2quat([repmat([0 0 1],length(bicyclePose),1) bicyclePose(:,3)]);

% plot(waypoints(:,1),waypoints(:,2),"kx-","MarkerSize",20);
% hold all

for i=1:400
    show(map');
    hold on

plotTransforms(bicycleTranslations(i,:),bicycleRot(i,:),'MeshFilePath','groundvehicle
.stl',"MeshColor","b");
    view(2);
    hold off;
    xlim([1,50]);
    ylim([1,50]);
    waitfor(vizRate);
end

plot(path(:,1),path(:,2));
```

5. Astar_path.m

```
%Anthony Chrabieh
%Bonus Problem
%A* Algorithm

clear all
clc
clf

load('ocpd.mat')

%Define Number of Nodes
xmax = 50;
ymax = 50;
MAP = zeros(xmax,ymax);

for i=1:51
    for j = 1:51
        if OCPD(i,j)==1
            MAP(i,j)=inf;
        end
    end
end

%Start and Goal
start = [2,2];
goal = [49,49];

%Nodes
%MAP = OCPD;

%To define objects, set their MAP(x,y) to inf
% MAP(40,15:25) = inf;
% MAP(35:40,15) = inf;
% MAP(35:40,25) = inf;

%To use Random Objects, uncomment this secion
% NumberOfObjects = 50;
% k = 0;
% while (k < NumberOfObjects)
%     length = max(3,randi(20));
%     x = randi(xmax-2*length)+length;
%     y = randi(ymax-2*length)+length;
%     direction = randi(4);
%     if (direction == 1)
%         x = x:x+length;
%     elseif (direction == 2)
%         x = x-length:x;
%     elseif (direction == 3)
%         y = y:y+length;
%     elseif (direction == 4)
%         y = y-length:y;
%     end
%     if (sum(isinf(MAP(x,y)))>0)
%         continue
%     end
```

```
%     MAP(x,y) = inf;
%     k = k + 1;
% end


%Heuristic Weight%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
weight = sqrt(2); %Try 1, 1.5, 2, 2.5
%Increasing weight makes the algorithm greedier, and likely to take a
%longer path, but with less computations.
%weight = 0 gives Djikstra algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Heuristic Map of all nodes
for x = 1:size(MAP,1)
    for y = 1:size(MAP,2)
        if(MAP(x,y)~=inf)
            H(x,y) = weight*norm(goal-[x,y]);
            G(x,y) = inf;
        end
    end
end

%Plotting%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% surf(MAP')
% colormap(gray)
% view(2)
%
% hold all
% plot(start(1),start(2),'s','MarkerFaceColor','b')
% plot(goal(1),goal(2),'s','MarkerFaceColor','m')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%initial conditions%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
G(start(1),start(2)) = 0;
F(start(1),start(2)) = H(start(1),start(2));

closedNodes = [];
openNodes = [start G(start(1),start(2)) F(start(1),start(2)) 0]; %[x y G F cameFrom]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Solve
solved = false;
while(~isempty(openNodes))

    pause(0.001)

    %find node from open set with smallest F value
    [A,I] = min(openNodes(:,4));

    %set current node
    current = openNodes(I,:);
%     plot(current(1),current(2),'o','color','g','MarkerFaceColor','g')

    %if goal is reached, break the loop
    if(current(1:2)==goal)
```

```
        closedNodes = [closedNodes;current];
        solved = true;
        break;
    end

    %remove current node from open set and add it to closed set
    openNodes(I,:) = [];
    closedNodes = [closedNodes;current];

    %for all neighbors of current node
    for x = current(1)-1:current(1)+1
        for y = current(2)-1:current(2)+1

            %if out of range skip
            if (x<1||x>xmax||y<1||y>ymax)
                continue
            end

            %if object skip
            if (isinf(MAP(x,y)))
                continue
            end

            %if current node skip
            if (x==current(1)&&y==current(2))
                continue
            end

            %if already in closed set skip
            skip = 0;
            for j = 1:size(closedNodes,1)
                if(x == closedNodes(j,1) && y==closedNodes(j,2))
                    skip = 1;
                    break;
                end
            end
            if(skip == 1)
                continue
            end

            A = [];
            %Check if already in open set
            if(~isempty(openNodes))
                for j = 1:size(openNodes,1)
                    if(x == openNodes(j,1) && y==openNodes(j,2))
                        A = j;
                        break;
                    end
                end
            end


            newG = G(current(1),current(2)) + round(norm([current(1)-x,current(2)-
y]),1);
```

```
            %if not in open set, add to open set
            if(isempty(A))
                G(x,y) = newG;
                newF = G(x,y) + H(x,y);
                newNode = [x y G(x,y) newF size(closedNodes,1)];
                openNodes = [openNodes; newNode];
%                 plot(x,y,'x','color','b')
                continue
            end

            %if no better path, skip
            if (newG >= G(x,y))
                continue
            end

            G(x,y) = newG;
            newF = newG + H(x,y);
            openNodes(A,3:5) = [newG newF size(closedNodes,1)];
        end
    end
end

if (solved)
    %Path plotting
    j = size(closedNodes,1);
    path = [];
    while(j > 0)
        x = closedNodes(j,1);
        y = closedNodes(j,2);
        j = closedNodes(j,5);
        path = [x,y;path];
    end

%     for j = 1:size(path,1)
%         plot(path(j,1),path(j,2),'x','color','r')
%         pause(0.01)
%     end
else
    disp('No Path Found')
end

for i=1:51
    for j=1:51
        if MAP(i,j)>1
            MAP(i,j)=1;
        end
    end
end

hold off
%show(binaryOccupancyMap(MAP))
MAP(2,2)=3;
MAP(49,49)=3;
surf(MAP')
view(2)
```

```
xlim([1,51])
ylim([1,51])
hold on
plot(path(:,1),path(:,2),'LineWidth',2.0,'Color','r')
title('A* Algorithm')
```

6. Visualize.m

```
clear v
% sz = size(timeline);
for i=1:8
    v(:,:) = timeline(i,:,:);
    surf(v);
    xlim([1,50])
    ylim([1,50])
    view(2)
    pause(2)
end
```