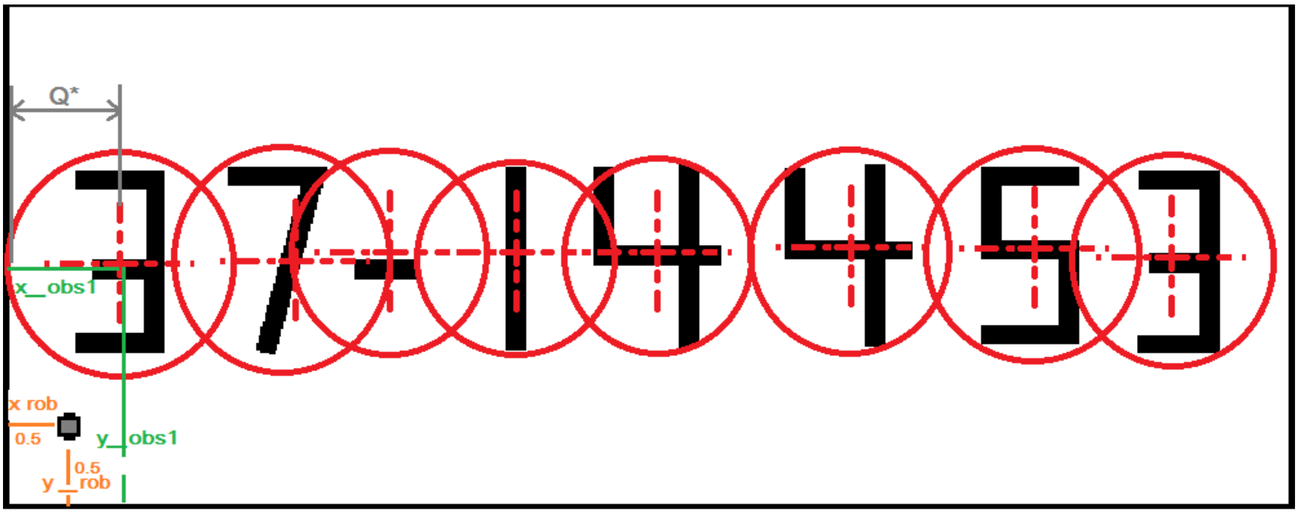
Made by : Saher

Autonomous Systems Milestone4

1-APF algorithm

as we imagined the map there



and for each obstacle the robot must calculate the repulsion forces according to it`s current position which will cause d_obs differs which will cause difference in repulsion forces every time as here

```
obstacles = [[1.3,2],[2.19,2],[3,2],[4.00,1.5],[4.00,1.9],[4.00,2],[5.2,2],[5.2,1.5],[5.2,1.7],[5.1,1.6],
             [5.0,1.8],[5.0,1.9],[5.15,1.9],[6.8,2],[6.8,1.6],[6.85,1.7],[6.65,1.6],[6.6,1.8],[6.75,1.9],
             [7.8,2],[8.1,2.2],[9.3,1.3],[9.5,2]]
def APF_Fn(Rob_pos,Goal_pos,APF_Param):
  global Fx_att
  global Fy_att
  global d_obs
  global Fx_rep
  global Fy_rep
  Fx_rep_val = 0
  Fy_rep_val = 0
  Fx_att_val = Fx_att.subs([(x_rob,Rob_pos[0]),(x_goal,Goal_pos[0])])
  Fy_att_val = Fy_att.subs([(y_rob,Rob_pos[1]),(y_goal,Goal_pos[1])])
  for obs in obstacles :
    if  Rob_pos[0] + 0.25 < obs[0] or Rob_pos[0] > obs[0] + 0.49:
      pass
    else:
        Obs_pos = obs
        d_obs_val = d_obs.subs([(x_rob,Rob_pos[0]),(y_rob,Rob_pos[1]),(x_obs,Obs_pos[0]),(y_obs,Obs_pos[1])])
        if d_obs_val<APF_Param[2]:
          Fx_rep_val = Fx_rep.subs([(x_rob,Rob_pos[0]),(y_rob,Rob_pos[1]),(x_obs,Obs_pos[0]),(y_obs,Obs_pos[1]),(d_obs,d_obs_val)])
          Fy_rep_val = Fy_rep.subs([(x_rob,Rob_pos[0]),(y_rob,Rob_pos[1]),(x_obs,Obs_pos[0]),(y_obs,Obs_pos[1]),(d_obs,d_obs_val)])
        else:
          Fx_rep_val = 0
          Fy_rep_val = 0
  Fx_net_val = Fx_att_val + Fx_rep_val
  Fy_net_val = Fy_att_val + Fy_rep_val
  F_xy_net = [Fx_net_val,Fy_net_val]
  return F_xy_net
# goes perfectly to (x,y) between charachers : (3.5,2.5) , (4.45,2.5), (5.8~6,2.5) ,(7.5,2.5)
# (8.5,2.5)
#Simulation While Loop
tau = rospy.get_param("~tau") #Sampling Time
rob_mass = rospy.get_param("~rob_mass") #Robot Mass (Turtlebot 3 Waffle_pi)
seen=[(0,0)]
```

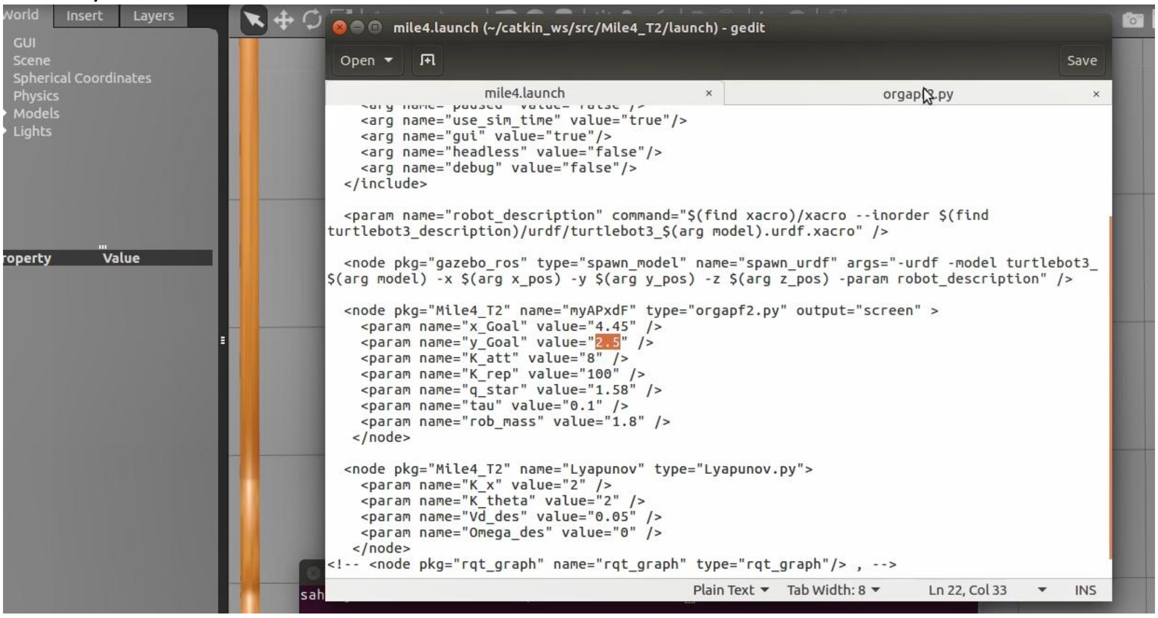
Made by : Saher

and a tricky line was add for more code optimization , if the robot passed the obstacle by 0.5 ~ 1/2 square , it neglect it`s repulsion , but if it will return near it , it will re-calculate it`s repulsion , but this trick was added to make robot more smarter and deals with obstacles in smart way

```
while 1 and not rospy.is_shutdown():
    if flag_cont == 1:
    #Get Robot Current Position and Velocity
    Rob_pos = [position[0],position[1],position[3]]
    Rob_vel = [velocity[0],velocity[5]]
    #Implement Artificial Potential Field
        F_xy_net = APF_Fn(Rob_pos,Goal_Pos,APF_Param)
    F_net = float(sqrt(F_xy_net[0]**2+F_xy_net[1]**2))
    F_net_direct = float(atan2(F_xy_net[1],F_xy_net[0]))
    if (Rob_pos[0] < F_xy_net[0]):
      pass
    #Calculate the desired robot position from the APF
    vel_c_x = vel_p_x + (F_xy_net[0]/rob_mass)*tau
    vel_c_y = vel_p_y + (F_xy_net[1]/rob_mass)*tau
    x_des = x_p + vel_c_x*tau
    y_des = y_p + vel_c_y*tau
    if (x_des,y_des)not in seen: # to avoid repeating forward and backward many times
        theta_des = F_net_direct
        Rob_pos_des = [x_des,y_des,theta_des]
    #Update the previous robot states for the next iteration
        vel_p_x = Rob_vel[0]*cos(Rob_pos[2])
        vel_p_y = Rob_vel[0]*sin(Rob_pos[2])
        x_p = Rob_pos[0]
        y_p = Rob_pos[1]
        #seen.append((x_des,y_des))
            flag_cont = 0
    else:
    Rob_pos_des = Rob_pos
    #goal Threashold station (becomes stationary when approaching goal vector by 0.2)
    if (sqrt((Rob_pos[0] - Goal_Pos[0] )**2+(Rob_pos[1] - Goal_Pos[1])**2) < 0.12):
    pass
    else :
      Des_Pos_msg.position.x = Rob_pos_des[0]
      Des_Pos_msg.position.y = Rob_pos_des[1]
      Des_Pos_msg.position.z = 0
      [qx_des, qy_des, qz_des, qw_des] = euler_to_quaternion(Rob_pos_des[2], 0, 0)
      Des_Pos_msg.orientation.x = qx_des
      Des_Pos_msg.orientation.y = qy_des
      Des_Pos_msg.orientation.z = qz_des
      Des_Pos_msg.orientation.w = qw_des
      pub1.publish(Des_Pos_msg) #Publish msg
      rate.sleep()      #Sleep with rate
#####################################################################################################
```

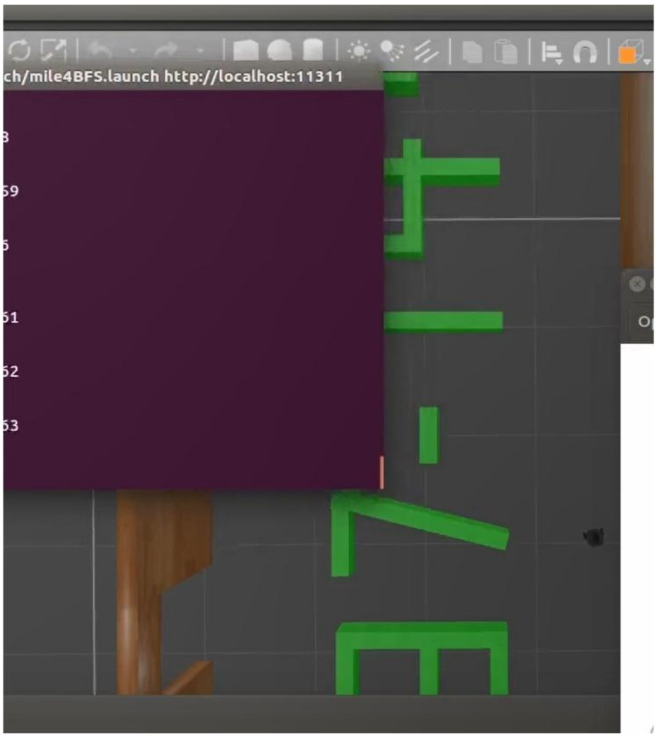
and the above part of the code is also as same as the introduced "laypanpouv" control but here a simple line was add to maintain the robot and protecting it from moving dummy moves "as rotating around it`s  axis which if it`s very near to obstacle it will hit it while it`s rotating around it`s axis , so when robot reaches goal or when goal threshold value > 0.12 it will try to maintain it`s goal without dummy rotations

Made by : Saher

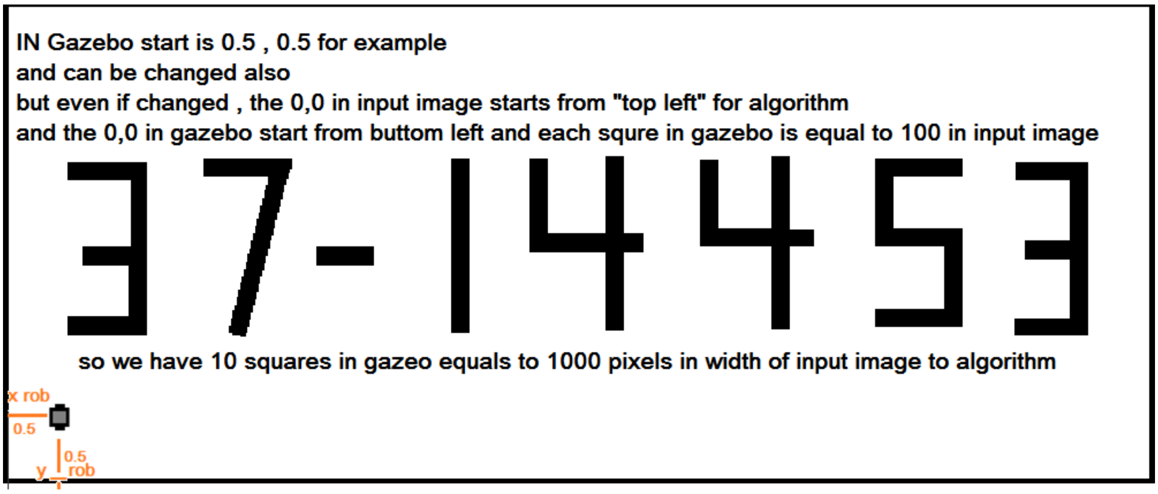


Made by : Saher
Here is launch file is the caller for the python file , as it`s also launches the map file (world) for the turtle bot , and in configuration there are entered parameters , which can be assigned in for of string and called from python file "rospy.getParam" , but first the node in python ile must be initialized as in next slide

2- BFS algorithm



as shown there the map of BFS the 0,0 of gazebo differs from 0,0 of the input that for the array of integers in BFS algorithm , as explained in BFS slides in narration of slides , and as shown below



so we have to compensate that one while dealing with start and goal point , for more complex goal from robot current position and pass through letters to reach it`s goals and every square in gazebo is equal to 100 pixels / movements in BFS algorithm , so this distance must be compensated from output of BFS path to go to goal controller that was used in both (A* , and BFS)

.

Made by : Saher

this part below extracts image into binary array

```
#np_img = np.array(img_inverted)
#np_img[np_img == 0] = 1 #not converted from 0 to 1 its converted to 0 and 255
# so above line remake the 0`s to 1 which is free space
# and the line below turns 255 into 0 which is obstacles
#np_img[np_img ==255] = 0
#np_img[np_img ==254] = 0
#matrix = np_img
#matrix_outx = np_img



img = cv2.imread('/home/saher/catkin_ws/src/Mile4_T2/src/Input.png')  #Read image
grayImage = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) #Convert RGB image to grayscale
ret, bw_img = cv2.threshold(grayImage,0,255,cv2.THRESH_BINARY) #Convert grayscale image to binary

bw_img = bw_img.astype(np.uint8)
matrix = bw_img
matrix_outx = bw_img
np_img = bw_img

np.set_printoptions(threshold=np.inf, linewidth=np.inf)  # turn off summarization, line-wrapping
with open('/home/saher/catkin_ws/src/Mile4_T2/src/output2D_Valid_Arry.txt', 'w') as f:
    f.write(np.array2string(np_img))
obstacles = ""
obstacles = ""
explored = ""
def is_obstacle(Cell_X,Cell_Y):
    if Cell_X <= 999 and Cell_Y <= 399 :
        valu = matrix[Cell_Y,Cell_X]
        if valu == 0:
            return True
    return False
def is_wall_cell(Cell_X,Cell_Y):
    if Cell_X == 0 and Cell_Y > 0:
        return True
    if Cell_X > 0 and Cell_Y == 0:
        return True
    if Cell_X == 0 and Cell_Y == 0:
        return True
    return False
def is_explored_cell(Cell_X,Cell_Y):
    if "i=" + str(Cell_X) in explored and "j=" + str(Cell_Y) in explored:
        return True
    return False
def add_explored_cell(Cel_X,Cel_Y):
    global explored
    str1 = "i=" + str(Cel_X)+ ","
    str2 = "j=" + str(Cel_Y)+ ","
    exploredx = str1+str2
    explored = explored + "".join(exploredx)
    return



def implement_BFS(start_point=[],goal_point=[]):
    global matrix
    var_queue = Queue()
    start_x = start_point[0]
    start_y = start_point[1]
    end_x = goal_point[0]
    end_y = goal_point[1]
    matrix[end_x,end_y] = 8 # <===== give a flag / Higlight the goal point (enha teba 8 fe west el 0s we el 1s)
    pth_list = bfs_helper(matrix, (start_x, start_y),(end_x,end_y))
    print("type now : " + str(type(pth_list)))
    return pth_list
```

this one is simple methods to make the process dealing with codes easier as possible and make it in a form to be understood well from any one read it

Made by : Saher

the part below implement a helper method for the BFS , and it`s not related to recursion , but it is a loop dependence , while the queue contains lists

```
def bfs_helper(grid, start,goalp):
    global matrix
    queue = collections.deque([[start]])
    goal = matrix[goalp[0],goalp[1]]
    height = len(matrix)
    width = len(matrix[0])
    print("Using BFS : " + str(width) + " X "  + str(height))
    print("Goal cell changed from 1 to value '8' for only Highlighting it => " + str(goal))
    seen = set([start])
    while queue:
        path = queue.popleft()
        x, y = path[-1]
        if grid[y][x] == goal:
            return path
        for x2, y2 in ((x,y+1) , (x,y-1), (x-1,y),(x+1,y),(x-1,y-1),(x+1,y+1),(x+1,y-1),(x-1,y+1)):
            if 0 <= x2 < width and 0 <= y2 < height and is_wall_cell(x2,y2)==False and is_obstacle(x2+24,y2+24)==False
            and is_obstacle(x2,y2+24)==False
            and is_obstacle(x2+24,y2)==False
            and is_obstacle(x2-24,y2)==False
            and is_obstacle(x2,y2-24)==False and is_obstacle(x2+24,y2-24)==False and
            is_obstacle(x2-24,y2+24)==False and (x2, y2) not in seen:
                queue.append(path + [(x2, y2)])
                seen.add((x2, y2))

#=========================== MAIN BFS CALL =========================
#++++++++++++++++++++++++++++
#====================================================================
#====================================================================
rospy.init_node("BFS_controller")
BFSstart = time.time() #  <========= capture time before execution
startx = rospy.get_param("~x_Start")
my_start = [int(startx), int(rospy.get_param("~y_Start"))] # giving start position i,j
<======================================    start point
my_goal = [int(rospy.get_param("~y_Goal")), int(rospy.get_param("~x_Goal"))]
#[40, 640] # giving end position j,i<=====================================    goal point
```

and them calculate time of process by capturing it before and after implementing and subtracting them and get the time taken to find the goal

```
path_arry_list = implement_BFS(my_start,my_goal) #<================================== needed BFS me
turtleBotpath_points =  path_arry_list
length = len(path_arry_list)
print('============== Given Path Using BFS ==============')
for ipj in range(length):
    x_path = path_arry_list[ipj][0]
    y_path = path_arry_list[ipj][1]
    matrix_outx[y_path,x_path] = 0
    print(" move in j : " +str(y_path) + " then " + "move in i : " +str(x_path) )
matrix_outx[matrix_outx == 4] = 1
matrix_outx[matrix_outx >4] = 130
BFSend = time.time()#  <========= capture time after execution
print ("Goal Reached ! , Time taken to find goal : " + str(BFSend-BFSstart))
np.set_printoptions(threshold=np.inf, linewidth=np.inf)
im = Image.fromarray(matrix_outx*255)
im.save("/home/saher/catkin_ws/src/Mile4_T2/src/Output_BFS.png")
#========================================BFS Writer=================================
with open('/home/saher/catkin_ws/src/Mile4_T2/src/path_out_BFS.txt', 'w') as f:
    f.write(np.array2string(matrix_outx))
print ("Path Drawn from start to goal which cell value is '8'")
print ("Path saved to 'Output_BFS.png' in same Dir of .py file If On Windows")
turtlebot_len = len(turtleBotpath_points)
x = 0.0
y = 0.0
theta = 0.0
def newOdom(msg):
```

the image above is code that "re-fine" the x,y from BFS and dividing them /100 and in Y it differs some how , this somehow is re-correction of the path out to be valid path in gazebo , and accuracy can depend on this numbers also

Made by : Saher

```
def newOdom(msg):
    global x
    global y
    global theta

    x = msg.pose.pose.position.x
    y = msg.pose.pose.position.y

    rot_q = msg.pose.pose.orientation
    (roll, pitch, theta) = euler_from_quaternion([rot_q.x, rot_q.y, rot_q.z, rot_q.w])
sub = rospy.Subscriber("/odom", Odometry, newOdom)
pub = rospy.Publisher("/cmd_vel", Twist, queue_size = 10)
speed = Twist()
r = rospy.Rate(10)
Krho = 0.08
Kalpha = 0.05
Kbeta = -0.05
goal = Point()
goal.x = 1.2
goal.y = 1.5
finalgoal_point = my_goal
final_turtle_x_desired = float(finalgoal_point[1]) / 100
final_turtle_y_desired = (380 - float(finalgoal_point[0])) / 100
#for itj in range(turtlebot_len):
print ("Final Desired is " + str(final_turtle_x_desired) + " y " + str(final_turtle_y_desired))
path_counter = 0
reached_bot = False
x = 0.0
r = rospy.Rate(10)
Krho = 0.2
Kalpha = 0.2
Kbeta = -0.1
goal = Point()
goal.x=  (50+float(turtleBotpath_points[path_counter][0])) / 100
goal.y=  (425-float(turtleBotpath_points[path_counter][1])) / 100
print ("Temp goal is " + str(goal.x) + " y " + str(goal.y))
while (sqrt((x - final_turtle_x_desired )**2+(y - final_turtle_y_desired)**2) > 0.3)  :
    if not rospy.is_shutdown() :
            if (sqrt((x - goal.x )**2+(y - goal.y)**2) < 0.3) :
        if (path_counter < turtlebot_len ) :
            path_counter = path_counter + 1
        elif (path_counter >= turtlebot_len ) :
            break
        goal.x = (10+float(turtleBotpath_points[path_counter][0])) / 100
        goal.y =(400-float(turtleBotpath_points[path_counter][1])) / 100
        print("i almost reached shifting goals to " + str(goal.x) + " " + str(goal.y))
            pass
            else :
        print ("Loop goal is " + str(goal.x) + " y " + str(goal.y))
        xd = goal.x -x
            yd = goal.y -y
            rho = sqrt((xd*xd)+(yd*yd))
            gamma = atan2(yd, xd)
            alpha = gamma - theta
            beta = 0 - gamma
            speed.linear.x = Krho*rho
            speed.angular.z = Kalpha*alpha + Kbeta*beta
            pub.publish(speed)
            r.sleep()
print ("Finished")
print ("Holding")
while not rospy.is_shutdown():
    xd = goal.x -x
    yd = goal.y -y
    rho = sqrt((xd*xd)+(yd*yd))
    gamma = atan2(yd, xd)
    alpha = gamma - theta
    beta = 0 - gamma
    speed.linear.x = Krho*rho
    speed.angular.z = Kalpha*alpha + Kbeta*beta
    pub.publish(speed)
    r.sleep()
```

the two images above represent the code of goto goal controller implemented and explained in narrated presentation.

Made by : Saher

3-The A*

Node

A node has a positioning value (eg. x, y), a reference to its parent and three 'scores' associated with it. These scores are how A* determines which nodes to consider first.

G score

The g score is the base score of the node and is simply the incremental cost of moving from the start node to this node.

$$g(n) = g(n.parent) + cost(n.parent, n)$$

$$cost(n_1, n_2) = \text{the movement cost from } n_1 \text{ to } n_2$$

H score - the heuristic

The heuristic is a computationally easy estimate of the distance between each node and the goal.

the image below is a pseudo code for A* algorithm

```
function A*(start, goal)
  open_list = set containing start
  closed_list = empty set
  start.g = 0
  start.f = start.g + heuristic(start, goal)
  while open_list is not empty
    current = open_list element with lowest f cost
    if current = goal
      return construct_path(goal) // path found
    remove current from open_list
    add current to closed_list
    for each neighbor in neighbors(current)
      if neighbor not in closed_list
        neighbor.f = neighbor.g + heuristic(neighbor, goal)
        if neighbor is not in open_list
          add neighbor to open_list
        else
          openneighbor = neighbor in open_list
          if neighbor.g < openneighbor.g
            openneighbor.g = neighbor.g
            openneighbor.parent = neighbor.parent
  return false // no path exists

function neighbors(node)
  neighbors = set of valid neighbors to node // check for obstacles here
  for each neighbor in neighbors
    if neighbor is diagonal
      neighbor.g = node.g + diagonal_cost // eg. 1.414 (pythagoras)
    else
      neighbor.g = node.g + normal_cost // eg. 1
    neighbor.parent = node
  return neighbors

function construct_path(node)
  path = set containing node
  while node.parent exists
    node = node.parent
    add node to path
  return path
```

Made by : Saher

the image below creates node class as explained in narrated presentation

```
class CELL():
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.from_the_start_dis = 0
        self.heuristic_distance = 0
        self.fn_obj = 0
    def __eq__(self, other):
        return self.position == other.position
```

the image below implements the pre-explained pseudo code

```
def perform_my_astar(maze, start, end):
    # Create start and end Cells
    start_node = CELL(None, start)
    start_node.from_the_start_dis =0
    start_node.heuristic_distance = 0
    start_node.fn_obj = 0
    end_node = CELL(None, end)
    end_node.from_the_start_dis = 0
    end_node.heuristic_distance = 0
    end_node.fn_obj = 0
    ro_saf_mr = 24
    # Initialize both open and closed list
    open_list = []
    closed_list = []
    # Add the start node
    open_list.append(start_node)
    # Loop until you find the end
    while len(open_list) > 0:
        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.fn_obj < current_node.fn_obj:
                current_node = item
                current_index = index
        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)
        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path
        # Generate Successors
```

Made by : Saher

```python
        # Generate Successors
        children = []
        for new_position in [(1, 1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (0, -1)]:
            # Get node position
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])
            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[len(maze)-1]) -1) or node_position[1] < 0:
                continue
            # Make sure walkable terrain
            nod_x = node_position[0]
            nod_y = node_position[1]
            if is_obstacle(nod_x+ro_saf_mr,nod_y+ro_saf_mr) == False and is_obstacle(nod_x+ro_saf_mr,nod_y) == False and is_obstacle(nod_x,nod_y+ro_saf_mr) == False
                                        # Create new node
                new_node = CELL(current_node, node_position)
                # Append
                children.append(new_node)
                # Loop through children
                for child in children:
                    if child in open_list :
                        continue
            # Child is on the closed list
                    for closed_child in closed_list:
                        if child == closed_child:
                            continue
            # Create the f, g, and h values
                    child.from_the_start_dis = current_node.from_the_start_dis + 1
                    child.heuristic_distance = ((child.position[0] - end_node.position[0]) ** 2) + ((child.position[1] - end_node.position[1]) ** 2)
                    child.fn_obj = child.from_the_start_dis + child.heuristic_distance
            # Child is already in the open list
                    for open_node in open_list:
                        if child == open_node and child.from_the_start_dis > open_node.from_the_start_dis:
                            # go to begining of loop again if dis from  start dis is bigger
                            continue
            # Add the child to the open list
                    if child not in open_list :
                        open_list.append(child)
##                  print(maze[node_position[0]][node_position[1]])
##                  print(maze[node_position[0]][node_position[1]])
##                  continue
            else :
                #print("obstacle at x " + str(nod_x) + "  " + str(nod_y))
                current_index +=1
```

and the goto goal controller as used in A* and path re-fining (correction) to be valid in gazebo according to the scaling ration (each square in gazebo = 100 movements)

```python
start = (x_Start, y_Start)
end = (x_Goal,y_Goal)

#obstacles are 0
# free path are 1 one s
i_pic = 0
j_pic = 0

#matrix[matrix == 0] = 0 #not converted from 0 to 1 its converted to 0 and 255
# so above line remake the 0`s to 1 which is free space
# and the line below turns 255 into 0 which is obstacles
matrix[matrix ==255] = 1
matrix[matrix ==254] = 1
mytime_started = time.time() #  <========= capture time before execution
print("Starting Algorithm")
np.set_printoptions(threshold=np.inf, linewidth=np.inf)  # turn off summarization, line-wrapping
with open('xx.txt', 'w') as f:
    f.write(np.array2string(matrix))
print("Please be patient if the goal in complex area [Finding Optimal Path]")
print("Working ...")
path = perform_my_astar(matrix, start, end)# search( matrix,1, start, end) #
for target_tuple in path:
    tuple_x = target_tuple[0]
    tuple_y = target_tuple[1]
    bw_img[int(tuple_y),int(tuple_x)] = 127
end_time = time.time()#  <========= capture time after execution
print ("Goal Reached ! , Time taken to find goal : " + str(end_time-mytime_started))
im = Image.fromarray(bw_img)
im.save("/home/saher/catkin_ws/src/Mile4_T2/src/Output_A_Star.png")
print("Press any key inside that Window of output map to starting moving TurtleBOT")
cv2.imshow("Window", bw_img)
cv2.destroyAllWindows() #Destroying present windows on screen
turtleBotpath_points =path
turtlebot_len = len(path)
x = 0.0
y = 0.0
theta = 0.0

def newOdom(msg):
```

Made by : Saher

the image below breaks when it`s reaches and after it holding holds the accurate goal point

```
goal.x=  (50+float(turtleBotpath_points[path_counter][0])) / 100
goal.y=  (400-float(turtleBotpath_points[path_counter][1])) / 100
print ("Temp goal is " + str(goal.x) + " y " + str(goal.y))

while (sqrt((x - final_turtle_x_desired )**2+(y - final_turtle_y_desired)**2) > 0.1)   :
    if not rospy.is_shutdown() :
            if (sqrt((x - goal.x )**2+(y - goal.y)**2) < 0.3) :
        if (path_counter < turtlebot_len ) :
             path_counter = path_counter + 1
             print("i will increase")
        elif (path_counter >= turtlebot_len ) :
             break
        if (path_counter >= turtlebot_len ) :
             goal.x = (10+float(turtleBotpath_points[path_counter-1][0])) / 100
             goal.y =(400-float(turtleBotpath_points[path_counter-1][1])) / 100
             break
        goal.x = (10+float(turtleBotpath_points[path_counter][0])) / 100
        goal.y =(400-float(turtleBotpath_points[path_counter][1])) / 100
        print("i almost reached shifting goals to " + str(goal.x) + " " + str(goal.y))
            pass
            else :
        print ("Loop goal is " + str(goal.x) + " y " + str(goal.y))
        xd = goal.x -x
            yd = goal.y -y
            rho = sqrt((xd*xd)+(yd*yd))
            gamma = atan2(yd, xd)
            alpha = gamma - theta
            beta = 0 - gamma
            speed.linear.x = Krho*rho
            speed.angular.z = Kalpha*alpha + Kbeta*beta
            pub.publish(speed)
            r.sleep()
print ("Finished My AStar")
 print ("Holding on Final Point")
while not rospy.is_shutdown():
    xd = goal.x -x
    yd = goal.y -y
    rho = sqrt((xd*xd)+(yd*yd))
    gamma = atan2(yd, xd)
    alpha = gamma - theta

    beta = 0 - gamma
    speed.linear.x = Krho*rho
    speed.angular.z = Kalpha*alpha + Kbeta*beta


    pub.publish(speed)
    r.sleep()
```