```
In [9]:
# python libraries
import numpy as np
import random
import matplotlib.pyplot as plt
import pandas as pd
```

```
In [10]:
ob_df = pd.read_csv('obstacles.csv')
print("obstacles dataframe csv read")
print(ob_df)
ob_df["#diameter"] =ob_df["#diameter"]/2
obstacles = list(zip(ob_df["#x"],ob_df["#y"],ob_df["#diameter"]))
print("format the obstacles list to use in programme")
print(obstacles)
```

```
obstacles dataframe csv read
       x      y  diameter
0 -0.285 -0.075      0.33
1  0.365 -0.295      0.27
2  0.205  0.155      0.15
format the obstacles list to use in programme
[(-0.285, -0.075, 0.165), (0.365, -0.295, 0.135), (0.205, 0.155, 0.075)]
```

```
In [11]:
# define the class graph with the startposition endposition and graphbounds

class Graph:

    def __init__(self, startpos, endpos,graphbound):
        self.startpos = startpos
        self.endpos = endpos
        self.xmin, self.ymin = graphbound[0]
        self.xmax, self.ymax = graphbound[1]

        self.vertices = [startpos]
        self.edges = []
        self.success = False

        self.vex2idx = {startpos: 0}
        self.neighbors = {0: []}
        self.distances = {0: 0.}

        self.sx = endpos[0] - startpos[0]
        self.sy = endpos[1] - startpos[1]

    def add_vex(self, pos):
        try:
            idx = self.vex2idx[pos]
        except:
            idx = len(self.vertices)
            self.vertices.append(pos)
            self.vex2idx[pos] = idx
            self.neighbors[idx] = []
        return idx

    def add_edge(self, idx1, idx2, cost):
        self.edges.append((idx1, idx2))
        self.neighbors[idx1].append((idx2, cost))
        self.neighbors[idx2].append((idx1, cost))

    def randomPosition(self):
        x = round(random.uniform(self.xmin,self.xmax),4)
        y = round(random.uniform(self.ymin,self.ymax),4)
        return x, y
```

```
In [12]:
def distance(node1, node2):
    # returs linear distance between node1 and node2
    return np.linalg.norm(np.array(node1) - np.array(node2))


def isinsideobstacle(node, obstacle):
    # check if the given node is inside the obstacles list
    # reyurs true if node is inside obstacles

    x, y = node
    obs = obstacle.copy()
    while len(obs) > 0:
        cx, cy, cr = obs.pop(0)
        d = (x - cx) ** 2 + (y - cy) ** 2
        if d <= cr ** 2:
            return True
        else:
            return False


def intersect(x1, x2, y1, y2, obstacles):
    # checkc if the line formed by node1(x1,y1) and node2(x2,y2) intersect with obstacles using interpo
lation
    # return true if the line cross with obstacles or tangent
    obs = obstacles.copy()
    while len(obs) > 0:
        cx, cy, cr = obs.pop(0)
        for i in range(0, 151):
            u = i / 100
            x = x1 * u + x2 * (1 - u)
            y = y1 * u + y2 * (1 - u)
            d = (x - cx) ** 2 + (y - cy) ** 2

            if d <= cr ** 2:
                return True
    return False


def nearest(G, vex, obstacles):
    # fine the nearest node
    Nvex = None
    Nidx = None
    minDist = float("inf")
    x1, y1 = vex
    for idx, v in enumerate(G.vertices):
        x2, y2 = v

        if intersect(x1, x2, y1, y2, obstacles):
            continue

        dist = distance(v, vex)
        if dist < minDist:
            minDist = dist
            Nidx = idx
            Nvex = v

    return Nvex, Nidx


def AddnewNode(randvex, nearvex, stepSize):
    #add new node in the direction of line with given stepsize

    dirn = np.array(randvex) - np.array(nearvex)
    length = np.linalg.norm(dirn)
    dirn = (dirn / length) * min(stepSize, length)

    newvex = (nearvex[0] + dirn[0], nearvex[1] + dirn[1])
    return newvex


def reconstruct_path(cameFrom, endpos, endid):
    # gives the path from start to end using parent tracing from cameFrom dictionaty input

    total_path = [[endid, endpos]]
    path_cost = 0
    while endid in cameFrom.keys():
        parent, node_coords = cameFrom[endid]
        total_path.insert(0, [parent, node_coords])  # insert the paernt node before the current node i
n total path list
        endid = parent
    return total_path
```

```
In [15]:
# defination of the RRT algorithm

def RRT(startpos, endpos, graphbound, obstacles, itr, stepSize):
    cameFrom = {}
    G = Graph(startpos, endpos, graphbound)

    for i in range(itr):

        rand_node = G.randomPosition()

        if isinsideobstacle(rand_node, obstacles):
            continue

        near_node, near_id = nearest(G, rand_node, obstacles)
        # continue loop if cant find nearest node
        if near_node is None:
            continue

        # add new node if its out of obstacles and nearest node is found
        new_node = AddnewNode(rand_node, near_node, stepSize)
        new_node_id = G.add_vex(new_node)

        cameFrom[new_node_id] = [near_id, near_node]

        dist = distance(new_node, near_node)
        G.add_edge(new_node_id, near_id, dist)

        dist = distance(new_node, G.endpos)
        if dist < 0.05:
            end_id = G.add_vex(G.endpos)
            G.add_edge(new_node_id, end_id, dist)
            G.success = True
            cameFrom[end_id] = [near_id, near_node]
            print('success')
            break  # break the loop when goal is reached

    return G, cameFrom,end_id  # return graph, parent list(cameFrom) and id of end node
```

In the following example RRT algorithm uses the 500 iteartions with stepsize 0.2 if you get an error : local variable 'end_id' referenced before assignment you should increase the number of iterations

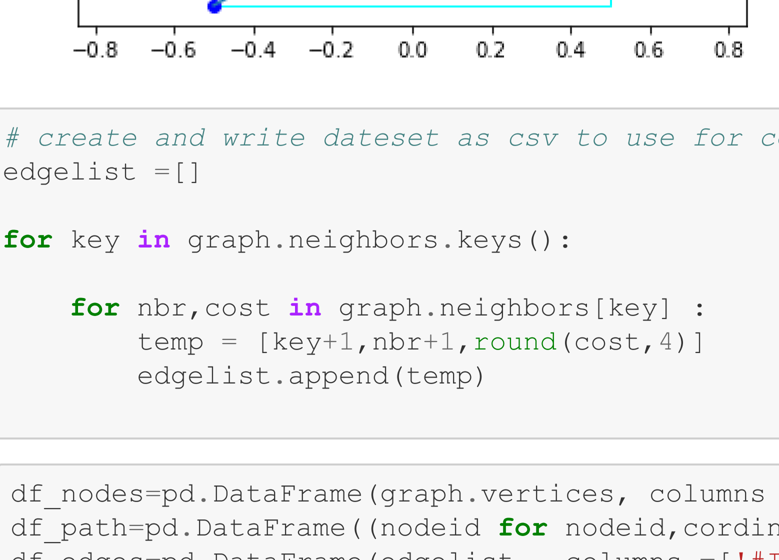
```
In [17]:
startpos = (-0.5,-0.5)
endpos = (0.5,0.5)
graphbound = [(-0.5,-0.5),(0.5,0.5)]
graph,cameFrom,endid = RRT(startpos, endpos, graphbound, obstacles, 500, 0.2)

print("end id of the end node is " , endid)
```

```
success
end id of the end node is  163
```

```
In [21]:
path = reconstruct_path(cameFrom,endpos, endid)
print("path from start node to reach end goal")
print(path)
```

```
path from start node to reach end goal
[[0, (-0.5, -0.5)], [1, (-0.32483178189914497, -0.40348007787315976)], [3, (-0.2247, -0.3756)], [4,
(-0.125, -0.214)], [5, (0.0613780480024623, -0.14145192474784835)], [10, (-0.02420662199285896, 0.039
311081040924595)], [19, (0.0838638778509185, 0.207598831823672)], [23, (0.036799999999999999, 0.277
2)], [24, (0.0883, 0.2999)], [26, (0.2829783388800334, 0.25407048580978764)], [45, (0.4434, 0.2425)],
[53, (0.4319, 0.2391)], [55, (0.3847, 0.2668)], [68, (0.4082, 0.329)], [101, (0.4595, 0.4239)], [163,
(0.5, 0.5)]]
```

```
In [19]:
# plot results

ax = plt.gca()
xmin,ymin = graphbound[0]
xmax,ymax = graphbound[1]
ax.set(xlim=(xmin-0.2, xmax+0.2), ylim=(ymin-0.2, ymax+0.2))
#plt.axis([bounds[0]-0.5, bounds[1]+0.5, bounds[0]-0.5, bounds[1]+0.5])
width = xmax-xmin
height = ymax-ymin
ax.add_patch(plt.Rectangle(graphbound[0],width, height,fill=False,color='cyan'))
plt.plot(xmin,ymin, 'bo')
plt.plot(xmax,ymax, 'ro')


for x1,y1,z1 in obstacles:
    circle = plt.Circle((x1, y1), z1, color='k')
    ax.add_patch(circle)

x_list = []
y_list = []

for nodeid, nodecords in path:
    x,y = nodecords
    x_list.append(x)
    y_list.append(y)

plt.plot(x_list,y_list)
plt.axis('equal')
plt.show()
```




```
In [ ]:
# create and write dateset as csv to use for coppeliasim
edgelist =[]

for key in graph.neighbors.keys():

    for nbr,cost in graph.neighbors[key] :
        temp = [key+1,nbr+1,round(cost,4)]
        edgelist.append(temp)
```

```
In [124]:
df_nodes=pd.DataFrame(graph.vertices, columns =['#x','#y'])
df_path=pd.DataFrame((nodeid for nodeid,cordinates in path) , index = None , columns =None).T + 1
df_edges=pd.DataFrame(edgelist,  columns =['#ID1','#ID2','#cost'])


df_nodes.index = np.arange(1, len(df_nodes)+1)
pd.DataFrame(df_nodes).to_csv('nodes.csv',float_format='%.4f')
pd.DataFrame(df_path).to_csv('path.csv',index=False,header=False,float_format='%.4f')
pd.DataFrame(df_edges).to_csv('edges.csv',index=False)
```