Supervised Learning Project Report

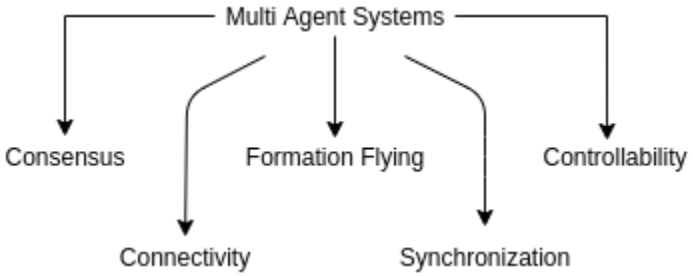
Student Name : Harshil Naik
Student Roll no: 190010030
Prof. Name : Prof. Shashi Ranjan Kumar
Date : 14th December2021

Topic : Multi-Agent Connectivity Maintenance

Motivation and Literature Review

I am really interested in the field of robotics and autonomous systems. Having done a lot of work in these fields as part of the student Technical team UMIC, I wanted to expand my reach further and take on newer challenges in the domain. Hence, Control and motion planning of multi agent systems was very interesting to me.

Thus, I started going through Survey papers which explained the application areas of multi agent systems in detail. Here, I got to know about applications like Multi Agent consensus algorithms, exploratory algorithms, formation flying, rendezvous, persistent monitoring and so on. Out of these, the Multi agent exploration interested me the most, as I also had some experience in High-level motion planning for quadrotors, hence it seemed like the natural extension. I then found a survey paper explaining about various approaches for multi agent search algorithms in detail. After discussing with sir, I weighed the pros and cons of each approach and finally selected a research paper that was best emulating the aspect of reality (limited sensing range, limited communication, decentralized nature and so on).



Link to the survey paper: “Survey on swarms”
I read through the paper in detail and understood their approach to tackling this problem. There were multiple layers to solving the problem. One of those layers was the maintenance of continuous connectivity, which meant that the graph formed by keeping the agents at vertices and assuming their communication links are edges, is completely connected (starting at any agent, one can always get a path to any other agent). This is known as continuous connectivity maintenance. I found well cited research papers on this topic as well.

Hence I realised that Multi Agent connectivity maintenance was the first step towards any application. One paper was cited almost everywhere, and contained a good amount of theoretical background in Graph Theory, from which this topic was derived.

Link to the paper for Connectivity Maintenance:
"A Passivity-Based Decentralized Strategy for Generalized Connectivity Maintenance"

Approach for this paper:
The adjacency matrix A consists of three gains (alpha, beta, and gamma). Each of those gains fulfills a particular requirement :

- Communication between two agents is possible only if they are within a certain sensing range of each other. (gamma)(**Hard requirement**)
- Agents keep a preferred interdistance between themselves for formation control (beta) (**Soft requirement**)
- Agents avoid collision with other agents (alpha) (**Hard requirement**)

Thus, I started to work towards these individual gains. The first requirement was about the agents always staying within the sensing range of each other to maintain overall graph connectivity. This was implemented in detail in another very well cited paper.

I decided to first implement this requirement and then eventually implement the other two requirements to finally finish the complete implementation.

The approach for this paper is described in detail below.
Hence, I decided to finalize this paper and implement it for this SLP.

The link to the paper :
"Decentralized Estimation and Control of Graph Connectivity in Mobile Sensor Networks"

Final Approach:

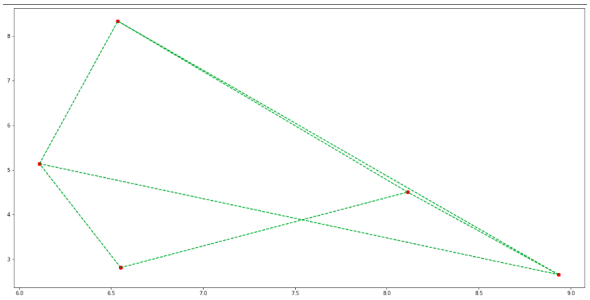
According to graph theory, a graph (Vertices and edges) is said to be completely connected if the Fiedler eigenvalue of its Laplacian matrix is positive. Despite the agent's movements coming from higher-level tasks like exploration targets, the graph should be connected always, to maintain communication between the agents.

Agent and environment model :

Given n mobile agents, their communication graph G and edge set E, the *adjacency* matrix $A \in \mathbb{R}^{n\times n}$ is defined as

$$A_{ij} = \begin{cases} A_{ji} = f(p^i, p^j), & i \text{ and } j \text{ are connected} \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

where $p^i, p^j \in \mathbb{R}^m$ are the positions of node i and j and we restrict $f(p^i, p^j) > 0$ to be a positive weighting function



Eigenvalue Estimation:

Centralized Approach:

Each agent starts with its own arbitrary estimate of the fiedler eigenvalue, which the central computing unit knows. It then runs a method called Power Iteration to estimate the fiedler eigenvector from the initial estimates of the agents.

> Given a square matrix Q and its spectrum $\mu_1 > \mu_2 \cdots > \mu_n$, *power iteration* is an established iterative method to compute its leading eigenvalue μ_1 and its associated eigenvector [19]. Now assume instead of μ_1, we are interested in its second largest eigenvalue μ_2. If we already know μ_1 and its associated eigenvector v_1, we can estimate μ_2 by running the power iteration on the deflated matrix
>
> $$\widetilde{Q} = Q - v_1 v_1^T. \tag{2}$$
>
> Specifically this iterative power iteration procedure is carried out in three steps. For a random initial vector w,
>
> 1) Deflate Q to get $\widetilde{Q}$, as in (2).
> 2) Power Iteration: $x \leftarrow \widetilde{Q}w$.
> 3) Renormalization: $w \leftarrow \frac{x}{\|x\|}$. Then go to step 3.

Assuming **x** to be the final estimated value of the fiedler eigenvector, a continuous time version of the Power Iteration consists of the following parts :

1) Deflation: $\dot{x} = -\text{Ave}(\{x^i\})\mathbf{1}$
2) Power Iteration: $\dot{x} = -Lx$
3) Renormalization: $\dot{x} = -(\text{Ave}(\{(x^i)^2\}) - 1)x$

Combining them, we get

$$\dot{x} = -k_1 \text{Ave}(\{x^i\})\mathbf{1} - k_2 Lx - k_3(\text{Ave}(\{(x^i)^2\}) - 1)x$$

Thus, we can get the Fiedler eigenvalue estimate from here after each step, and check whether it is positive or not.

The derivative of the eigenvector estimate here was calculated from the Power iteration using a centralized approach, by assuming that we can find the average terms using all the estimates. But in reality, it is always better to have a decentralized algorithm, so that there is no vulnerable central processing unit. Thus, we choose to implement a decentralised approach next.

Decentralized Approach:

The decentralised approach assumes that each agent has a 1-hop communication model, which means that each agent can share its own information with its immediate neighbours which lie within its sensing range. The main term that need to be decentrally computed in the previous equations were the “average” terms, that need to be computed just using the immediate sensed neighbour information.

To convert the previously described Power interaction into its decentralised form, we use the PI Average consensus algorithm, which is as follows :

$$\dot{y}^i = \gamma(\alpha^i - y^i) - K_P \sum_{j \in \mathcal{N}^i} [y^i - y^j] + K_I \sum_{j \in \mathcal{N}^i} [w^i - w^j]$$
$$\dot{w}^i = - K_I \sum_{j \in \mathcal{N}^i} [y^i - y^j].$$

Where N_i is the set of immediate sensed neighbours, y_i is the estimate of the Avg(x_i) term.

In the decentralized implementation of (3), agent i maintains a scalar x^i (which converges to the i-th component of the eigenvector $\tilde{v}_2$) and four consensus estimator states $\{y^{i,1}, w^{i,1}, y^{i,2}, w^{i,2}\}$ ($y^{i,1}$ and $y^{i,2}$ are agent i's estimates for Ave($\{x^i\}$) and Ave($\{(x^i)^2\}$) respectively) and receives from communication its neighbors' $\{x^j, y^{j,1}, w^{j,1}, y^{j,2}, w^{j,2}\}$ for all $j \in \mathcal{N}^i$. Noticing $-Lv_2 = \lambda_2 v_2$, agent i can estimate λ_2 through

$$\lambda_2^i = -\frac{\sum_{j \in \mathcal{N}^i} L_{ij} x^j}{x^i} \tag{9}$$

Thus, we can now get the Fiedler eigenvalue and the Fiedler eigenvector of the graph in a decentralized manner.

Control Input to maintain connectivity:

Now, we are able to estimate the Fiedler eigenvalue at each timestep. But suppose we have a high-level control input as well - such as a specific target position for one of the agents. Then, the other agents need to move in such a way so that connectivity is still maintained. Thus, we need to calculate the control input for such a situation explicitly. The control input for each agent **k** in such a situation is defined as :

$$u^k = \dot{p}^k = \frac{\partial \lambda_2}{\partial p^k} = \hat{v}_2^T \frac{\partial L}{\partial p^k} \hat{v}_2.$$

$$u^k = \tilde{v}_2^T \frac{\partial L}{\partial p^k} \tilde{v}_2 = \sum_{(i,j) \in E} \frac{\partial A_{ij}}{\partial p^k} (\tilde{v}_2^i - \tilde{v}_2^j)^2. \tag{15}$$

Since we have defined $A_{ij} = e^{-\|p^i - p^j\|_2^2 / 2\sigma^2}$, we can compute

$$\frac{\partial A_{ij}}{\partial p^i} = -A_{ij}(p^i - p^j)/\sigma^2 \qquad i \neq j \tag{16}$$
$$\frac{\partial A_{ij}}{\partial p^j} = A_{ij}(p^i - p^j)/\sigma^2 \qquad i \neq j \tag{17}$$
$$\frac{\partial A_{ii}}{\partial p^i} = 0 \tag{18}$$
$$\frac{\partial A_{ij}}{\partial p^k} = 0 \qquad k \neq i, j. \tag{19}$$

Thus,

$$u^k = \sum_{j \text{ such that } (k,j) \in E} -A_{kj}(x^k - x^j)^2 \frac{p^k - p^j}{\sigma^2}.$$

Where x_j is each of the neighbouring agents' estimates of the fiedler eigenvector, and p_j is their positions. Sigma is a parameter.

Thus, each agent moves according to uk to maintain connectivity in the graph.

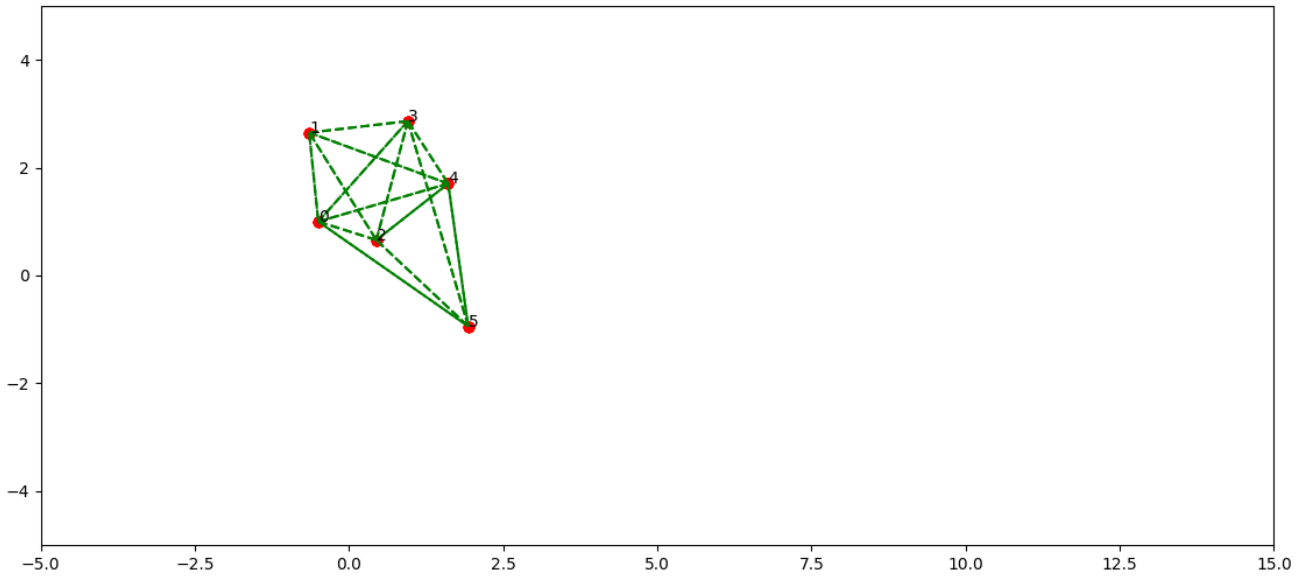
Code

The code has been written using OOP classes in python. I made a separate class for a general "**agent**" which was used to instantiate multiple agents in the **main** class, and to perform the overall code calculations for simulating the environment and handling I/O.

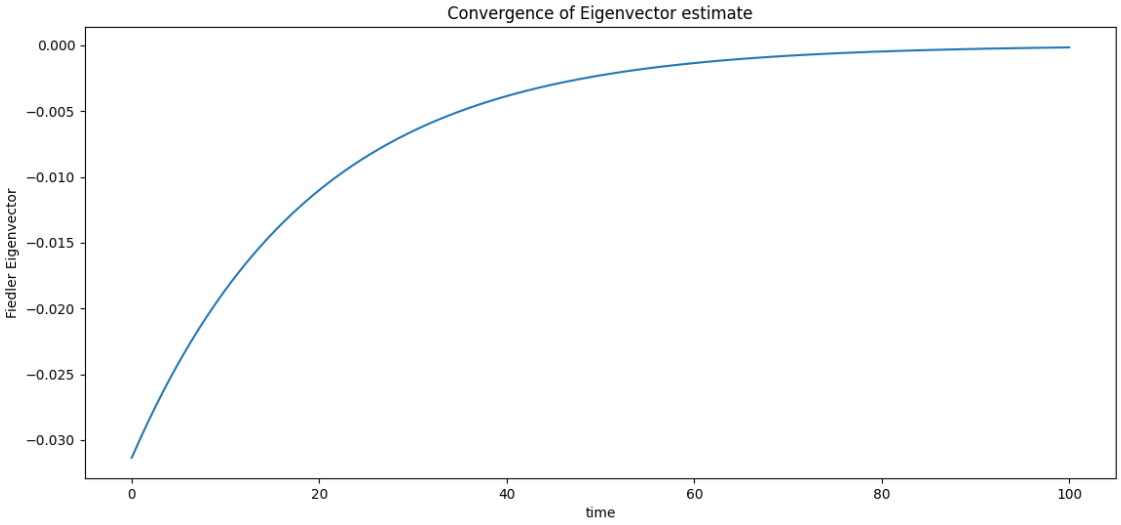
The complete code is given in the Appendix.

Final results

The following is a snippet of the final implementation. The dotted edges are between agents which can communicate with each other.



As you can see from the below graph of the eigenvector estimate vs. time, the estimate converges pretty quickly because of the PI Average consensus estimate algorithm. Hence, the graph stays connected even in the situation of external movement of one of the agents.



Future Plans

I have no intention of stopping at just these results. I plan to implement the remaining two requirements of the main connectivity paper to account for inter agent collision and inter agent distance.

Appendix (Code)

```
import sys
import os
import numpy as np
import time
import math
import matplotlib.pyplot as plt
import matplotlib

class agent:
   def __init__(self, agentnum, xcoord, ycoord, obstacles):
       self.xcoord = xcoord            # X coordinates to simulate environment
       self.ycoord = ycoord            # Y coordinates to simulate environment
       self.agentnum = agentnum        # Current agent number
       self.obstacles = obstacles      # Obstacle data for handling (Simulating the environment)
       self.A = np.zeros((len(self.xcoord), len(self.ycoord)), dtype=float)    # Graph Adjacency array
       self.N = []                     # Neighbours list
       self.ND = []                    # Neighbours Data
       self.sensrange = 4.0            # Sensing Range
       self.agentrange = 4.0           # Min. Distance from Agents
       self.x = 5                      # Arbitrary Estimate of eigenvector
       self.y1 = np.random.random()    #Arbitrary estimate for PI Consensus
       self.w1 = np.random.random()    #Arbitrary estimate for PI Consensus
       self.y2 = np.random.random()    #Arbitrary estimate for PI Consensus
       self.w2 = np.random.random()    #Arbitrary estimate for PI Consensus
       self.Kp = 500                   # PI Consensus gain
       self.Ki = 200.0                 # PI Consensus gain
       self.sigma = 10.0               # PI Consensus paramter

   def getWeights(self):
       # ------------------------------------------------------------------------------------------------------------
       # Update the adjacency matrix A
       # ------------------------------------------------------------------------------------------------------------
       for i in range(len(self.xcoord)):
           for j in range(len(self.xcoord)):
               if math.sqrt((self.xcoord[i] - self.xcoord[j])**2 + (self.ycoord[i] - self.ycoord[j])**2) < 4.0:
                   if j!=i:
                       self.A[i,j] = math.exp(-1*((self.xcoord[i] - self.xcoord[j])**2 + (self.ycoord[i] - self.ycoord[j])**2)/(2*self.sigma**2))
               else:
                   self.A[i,j] = 0.0

   def getNeighbours(self, neighboursdata):
       # ------------------------------------------------------------------------------------------------------------
       # Update Neighbours data based on sensor model
       # ------------------------------------------------------------------------------------------------------------
       for j in range(len(self.xcoord)):
           if j!= self.agentnum:
               dist = math.sqrt((self.xcoord[self.agentnum] - self.xcoord[j])**2 + (self.ycoord[self.agentnum]-self.ycoord[j])**2)
               if dist < self.sensrange:
                   self.N.append(j)
                   self.ND.append(neighboursdata[j])

   def PIaverageconsensus(self, agentnum, L, N, ND, xi, y1, y2, w1, w2):
       # ------------------------------------------------------------------------------------------------------------
       # PI Average consensus estimator function. Gives an estimate as the output.
       # ------------------------------------------------------------------------------------------------------------
       xini = xi
       xfin = xi + 100
       delT = 0.1
       k1, k2, k3 = 0.01,0.01,0.01
       while (abs(xfin - xini)) > 0.1:
           xdot = -k1*y1 - k2*L*xfin - k3*(y2 - 1)*xfin
           xfin += min(xdot[agentnum])*delT
```

```
            sum1, sum2, sum3, sum4 = 0.0,0.0,0.0,0.0
            for i in range(len(N)):
                sum1 += (y1 - ND[i][1])
                sum2 += (w1 - ND[i][2])
                sum3 += (y2 - ND[i][3])
                sum4 += (w2 - ND[i][4])
            w1dot = -self.Ki*sum1
            w2dot = -self.Ki*sum3
            w1 += w1dot*delT
            w2 += w2dot*delT
            y1dot = -self.Kp*sum1 + self.Ki*sum2
            y2dot = -self.Kp*sum3 + self.Ki*sum4
            y1 += y1dot*delT
            y2 += y2dot*delT
        return xfin

    def controlLaw(self, Ak, xfin, N, ND, x, y):
        # ------------------------------------------------------------------------------------------------------------
        # Gives the control output u that maintains connectivity for the agent based on the adjacency matrix and the agent's estimate of the
eigenvector
        # ------------------------------------------------------------------------------------------------------------
        uk = np.array([0.0,0.0])
        for i in range(len(N)):
            xi, yi = N[i][0], N[i][1]
            uk += -Ak[i]*((ND[i][0] - xfin)**2)*(np.array([x, y]) - np.array([xi, yi]))/(self.sigma**2)
        return uk


class main:
    def __init__(self):
        self.numagents = 6
        self.numobstacles = 10
        self.xcoord = [-2.0,-1.0,0.0,1.0,2.0,2.0]
        self.ycoord = [1.0, 3.0, 0.0, 3.0, 2.0, -1.0]
        self.obstacles = [[np.random.uniform(-7,7), np.random.uniform(-7,7)] for i in range(self.numobstacles)]
        self.neighboursdata = [[] for _ in range(self.numagents)]
        self.agents = []
        self.sigma = 0.00008

    def mainthread(self):
        # ------------------------------------------------------------------------------------------------------------
        # Simulates the environment and handles the main thread processing.
        # ------------------------------------------------------------------------------------------------------------
        avg1 = 0.0
        avg2 = 0.0
        xwei = []
        for i in range(self.numagents):             # Get the first initial estimates for variables going to be used ahead.
            agentnum = i
            agentclass = agent(agentnum, self.xcoord, self.ycoord, self.obstacles)
            self.agents.append(agentclass)
            self.neighboursdata[i].extend([agentclass.x, agentclass.y1, agentclass.w1, agentclass.y2, agentclass.w2])
            agentclass.getWeights()
            avg1 += agentclass.x
            avg2 += (agentclass.x)**2
            xwei.append(agentclass.x)

        x = np.zeros(self.numagents)                # Estimated eigenvector
        k1, k2, k3 = 0.12,0.02,0.004
        xp = np.linspace(0,100, 500)
        yp = []
        plt.figure(figsize=(20,10))
        ax = plt.axes()
        plt.xlim((-20,20))
        plt.ylim((-10,10))
        for k in range(500):
            self.xcoord[0] += 0.5                   # One agent is moving according to an external law, and other agents are maintaining connectivity
accordingly.
            for i in range(self.numagents):
                agentclass = self.agents[i]
                agentclass.getWeights()
                agentclass.getNeighbours(self.neighboursdata)
                d = (agentclass.A).dot(np.transpose(np.ones(self.numagents)))
                D = np.diag(d)
                L = D - agentclass.A
                avg1 = 0.0
                avg2 = 0.0
```

```
            for p in range(self.numagents):
                avg1 += xwei[p]
                avg2 += xwei[p]**2
            xdot = -k1*(avg1/self.numagents)*np.ones(self.numagents) - k2*L*x - k3*(avg2/self.numagents - 1)*x
            x += xdot[i]*0.015
            xwei[i] = x[i]
            uk = np.array([0.0,0.0])
            for j in agentclass.N:
                uk += -agentclass.A[i][j]*((x[i] - x[j])**2)*(np.array([self.xcoord[i], self.ycoord[i]]) - np.array([self.xcoord[j],
self.ycoord[j]]))/(self.sigma**2)
            if i==0:
                uk[0], uk[1] = 0.0,0.0
            if i != 0:                          # Update the graph only for those points using the control input that are not being controlled.
                self.xcoord[i] += uk[0]*0.1
                self.ycoord[i] += uk[1]*0.1
            plt.clf()
            for i in range(len(self.xcoord)):   # Plotting code
                for j in range(len(self.ycoord)):
                    xd = [self.xcoord[int(i)], self.xcoord[int(j)]]
                    yd = [self.ycoord[int(i)], self.ycoord[int(j)]]
                    if math.sqrt((xd[0]-xd[1])**2 + (yd[0]-yd[1])**2) < 4.0:
                        plt.plot(xd,yd, '--',color='g')
                    plt.scatter(self.xcoord[int(i)], self.ycoord[int(i)], marker='o', color = 'r')
                plt.annotate("{}".format(i), (self.xcoord[i], self.ycoord[i]))
            plt.xlim([-5, 15])
            plt.ylim([-5, 5])
            plt.show(block=False)
            plt.pause(0.1)
        avg1 = 0.0
        avg2 = 0.0
        yp.append(max(x))
    # plt.plot(xp, yp)
    # plt.xlabel("time")
    # plt.ylabel("Fiedler Eigenvector")
    # plt.title("Convergence of Eigenvector estimate")
    plt.show()

if __name__=='__main__':
  mainthread = main()
  mainthread.mainthread()
```