ROS Implementation of Global and Robust Formation-Shape Stabilization of Relative Sensing Networks

Shengdong Liu

March 21, 2016

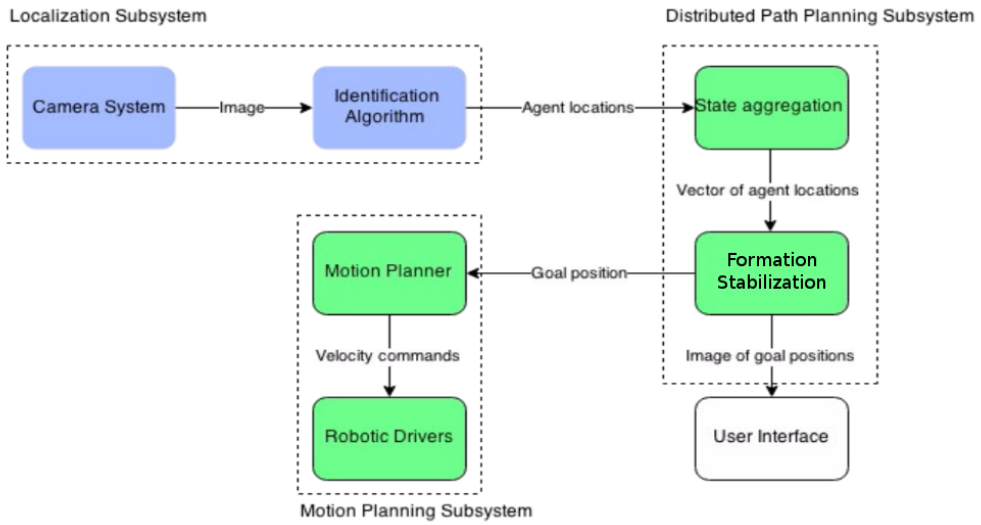
Abstract

As the Internet of things continues to grow and autonomous vehicles technologies continues to advance, a means to coordinate a network of autonomous vehicles becoming increasingly important. This work aims to implements and test a global and robust formation stabilization algorithm[1], by Prof. Jorge Cortés at University of California San Diego, for a network autonomous agents. This algorithm was implemented in Python and tested using ROS (Robotic Operating System) framework and Turtlesim, (simulation environment for TurtleBots). The simulation results shows that agents consistently stabilize in the given formation which can be translated and rotated.

1 Big Picture

As robotics technology advance, we are seeing an increasing the number of autonomous robots in our daily lives. This boom in robots calls for inter-robot communication and coordination to complete tasks in groups. The Multi-Agent Robotics Lab (MURO)[2] in UCSD, led by Professor Cortés and Martínez, has been incubating a testbed to test potential multi-agent control algorithms for deployment. The lab currently possess ten TurtleBots[3] platforms as a multi-agent robotic network. The open source ROS (Robot Operating System)[4] is used as the software platform for its publisher-subscriber model to allow for inter-robot communication and coordination. The objective of the ROS TurtleBot project is to provide a stable system to which distributed control systems can be readily deployed for testing and validation on hardware.

The current system set up for the testbed consists of three subsystems. The localization subsystem provided the position and orientation of each of deployed TurtleBots through two overhead cameras that send video stream to a laptop as the central station. The distributed path planning system applies multi-agent control algorithms and uses the video steams to calculate the goal position of each TurtleBot. The motions planning subsystem on each TurtleBot then uses the goal position to create a path and send velocity commands to the mobile base for navigation. Since the system is implement in ROS, information can be easily passed on by means of publishers and subscribers between the subsystems. Visualizations of the multi-agent control algorithm are created and displayed the users through the user interface for monitoring the correctness and effectiveness of the multi-agent control algorithm being tested.



2 Personal Contribution

This quarter, I worked on a part of distributed path planning subsystem, which calculates the goal position of each TurtleBot based on their current positions. I implemented, in Python, an formation stabilization algorithm proposed by Prof. Cortés. In his paper "Global and robust formation-shape stabilization of relative sensing networks", Prof Cortés describes a simple distributed algorithm that achieves global stabilization of formations for relative sensing networks in arbitrary dimensions with fixed topology. Assuming the network runs an initialization procedure to equally orient all agent reference frames, convergence to the desired formation shape is guaranteed even in partially asynchronous settings[1]. After implementing the algorithm, I wrapped algorithm in a ROS node in order to forward the goal positions to the motion planner. I tested my implementation in TurtleSim[5], a simulation tool for TurtleBots, and found that the TurtleBots stabilizes in the given formation which can be translated and rotated.

3 Preliminaries

In this section, we introduce necessary notations to understand the formation stabilization algorithm and summarize the update rule used for the implementation.

For the details and proofs related to the algorithm, please refer to the paper "Global and robust formation-shape stabilization of relative sensing networks."[1]

3.1 Notations

let n denotes the number of agents.
let d denotes the dimension of Euclidean space.
let G denotes a weighted digraph of all the agents.
let $D_{out}(G)$ denotes the $n \times n$ weighted out-degree matrix of the digraph G.
let $A(G)$ denotes the $n \times n$ weighted adjacency matrix with the following properties: for $i, j \in \{1, ..., n\}$, the entry $a_{ij} > 0$ if (i, j) is an edge of G, and $a_{ij} = 0$ otherwise.
let $L(G) = D_{out}(G) - A(G)$ denotes the graph Laplacian of the weighted digraph G.
let I_d denotes the $d \times d$ identity matrix.
let Z^* denotes the $n \times d$ matrix that represent the relative coordinates of the desired formation.

3.2 Update Rule

The update rule (equation 11 on the formation stabilization paper[1]) for the position of each agent in global frame is:

$$p_i(l+1) = (1-h)p_i(l) + h\frac{1}{d_i}(\sum_{j \neq i} a_{ij}p_j(l) + Rb_i) \tag{1}$$

Such that :

- l is the current time frame.
- $p_i(l)$ is the position of agent i in the current time frame.
- h is the rate in the range [0,1] at which the agents converge to the formation.
- d_i is the ith diagonal element of $D_{out}(G)$
- a_{ij} is the element on the i-th row and j-th column of $A(G)$.
- R is a $d \times d$ rotation matrix.
- b is a $n \times d$ matrix define by equation 8 on the formation stabilization paper[1] as:

$$b = (b_1, ..., b_n) = (L(G) \otimes I_d)Z^* \tag{2}$$

4 Methodology

In this section, we go over the rationale for test parameters used, the code implementation, the test setup and test results.

4.1 Rationale for Test Parameters

We set $d = 2$ because TurtleBots are ground vehicles, so the test should be conducted in 2D. (Note: I renamed d to dim in my code to avoid confusing it with the variable d_i in the algorithm)
We set $n = 4$ because 4 agents are enough to form simple formations, yet not too complicated to test and debug.
We set $D_{out}(G) = I_4$ to achieve a full connected graph where any agent is affected by all the other agents.
We set

$$A(G) = \begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/3 & 0 & 1/3 & 1/3 \\ 1/3 & 1/3 & 0 & 1/3 \\ 1/3 & 1/3 & 1/3 & 0 \end{bmatrix}$$

so each agent is equally affected by all the other agents.
We set

$$Z_1 = \begin{bmatrix} 0 & 0 \\ 0 & 3 \\ 3 & 3 \\ 3 & 0 \end{bmatrix} \quad Z_2 = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}$$

which produce a square formation and line formation respectively.
We set h to 0.25 for a smooth yet timely convergence.
Lastly, We set

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

as the 2D rotation matrix[6].

4.2 Implementation

The ROS framework support C++ and Python languages. The latter was chosen for this work for the ease of implementation and for its optimization for vector and matrix operations.

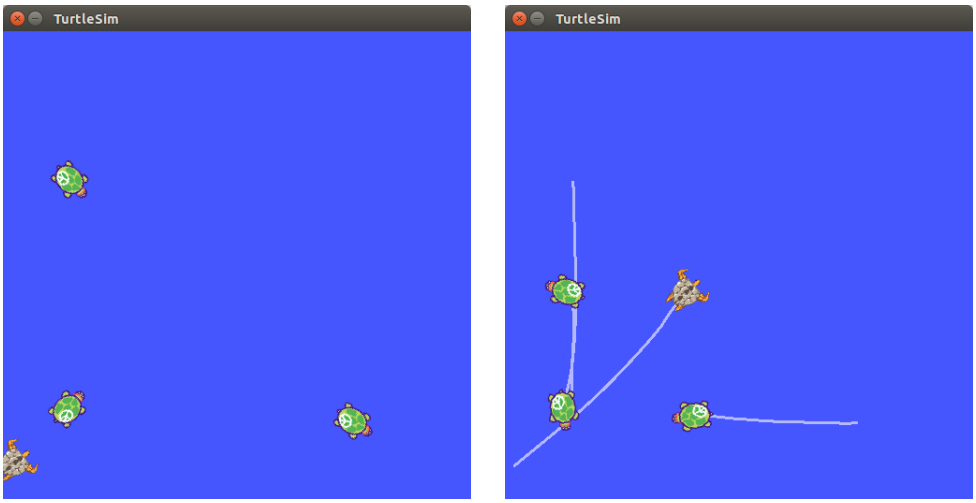
The following three blocks of code calculate the b matrix (equation 2), define the 2D rotation matrix, and implements the formation update rule (equation 1) for the agents to stabilize into the desired formation.

```
##########################################################
# finds the b Matrix in the JOR formula given
# the formation z
##########################################################
def findB(z):
    # vectorize the z matrix
    z = z.reshape(1, n*dim, order='F')

    # kron = (D_out(G) - A(G)) ⊗ I_d
    kron = np.kron(np.identity(dim), (d - a))

    # b = (kron)Z* and convert b matrix back to n × d
    b = (np.inner(kron, z)).reshape(n,dim, order='F')

    return b

#########################################################
# 2D rotation matrix for a given theta
#########################################################
def R(theta):
    return np.array([[ np.cos(theta), np.sin(theta)],
```

```
                [-np.sin(theta), np.cos(theta)]])

#####################################################
# Algorithm for Updating the Goal Positions
# ps - nxd matrix representing positions of all agents
# b  - b matrix in JOR algorithm
# theta - angle of the matrix
# h - convergence rate [0-1]
#     0 -> never
#     1 -> instantly
#####################################################
def updateGoalPoses(ps, b, theta=0., h=.25):
    # declare an empty matrix of dim columns
    toReturn = np.empty([0,dim])

    # calculate Rb
    rb = np.dot(b, R(theta))

    # loop through position of each agent
    for i in range(n):

        # initialize a vector of zero
        summ = np.zeros(dim)

        # summ = sum_{j!=i} a_ij p_j(l)
        summ = np.dot(a[i], ps)

        # p_i(l+1) = (1-h)p_i(l) + h*(1/d_i)(summ + Rb_i)
        p_new = (1.0-h)*ps[i] + (h/d[i][i])*(summ + rb[i])

        # stack the positions together to build the updated pos matrix
        toReturn = np.append(toReturn, [p_new], axis=0)

    return toReturn
```

4.3 Test Set Up

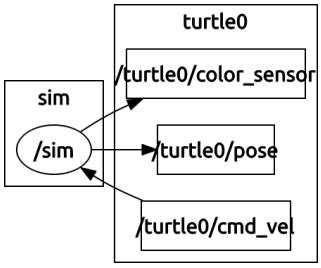
The testing was conducted in Turtlesim, a 2D TurtleBot simulator, using ROS framework. The formation stabilization implementation was wrapped in a ROS node name "formation_stailization". Turtlesim simulates a 11m by 11m square of space and each turtle on the simulator represents a TurtleBot. In the beginning of our simulation four turtles are spawned at random positions and are should stabilize at a desired formation like the images shown below.



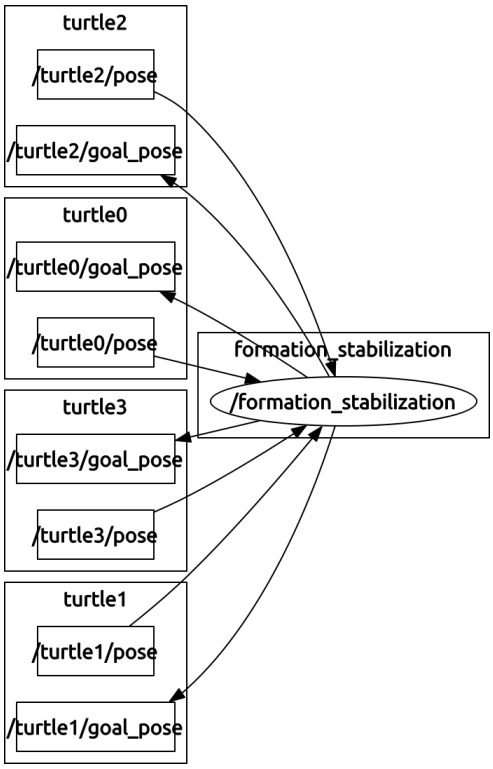
To understand what is needed for the simulation, we need to take a look at what ROS nodes are running and how they interact with each other. There are four types of nodes that are running while the simulation takes place:

- **sim** node is responsible for spawning turtles and simulating their positions and movements.
- **formation_stabilization** node is the wrapper node for our update algorithm.
- **poseConv** node converts the position in turtle sim to the type of position that actual TurtleBots use.
- **first_turn_then_move** node is a basic movement algorithm we use to drive the TurtleBot.

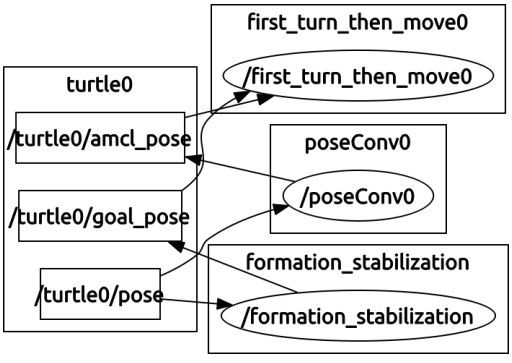
The next few graphs shows the interaction between the aformentioned ROS nodes.



In this graph, we see the **sim** node spawns turtle0 and publish the simulated position to "/turtle0/pose". **sim** node also subscribes to "/turtle0/cmd_vel" to simulate turtle0's movement using the command velocity.

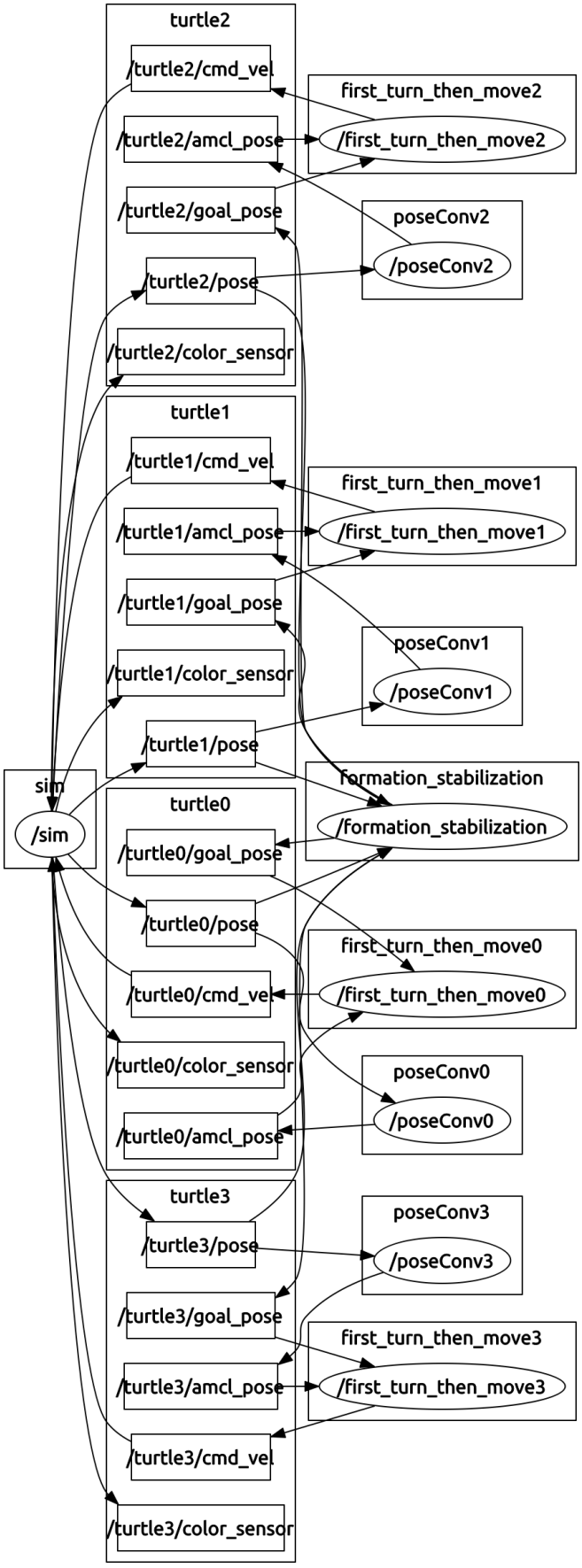


This graph shows the **formation_stabilization** node uses the current positions of the turtles to compute their goal positions and then publishes that information.



This graph shows that each turtle has its **poseConv** node to convert its simulated position to the format that TurtleBots use, and then its **first_turn_then_move** node uses that current position and navigate the turtle toward its goal position.

Now, putting everything together, we have the all the nodes and topics that runs in the simulation:



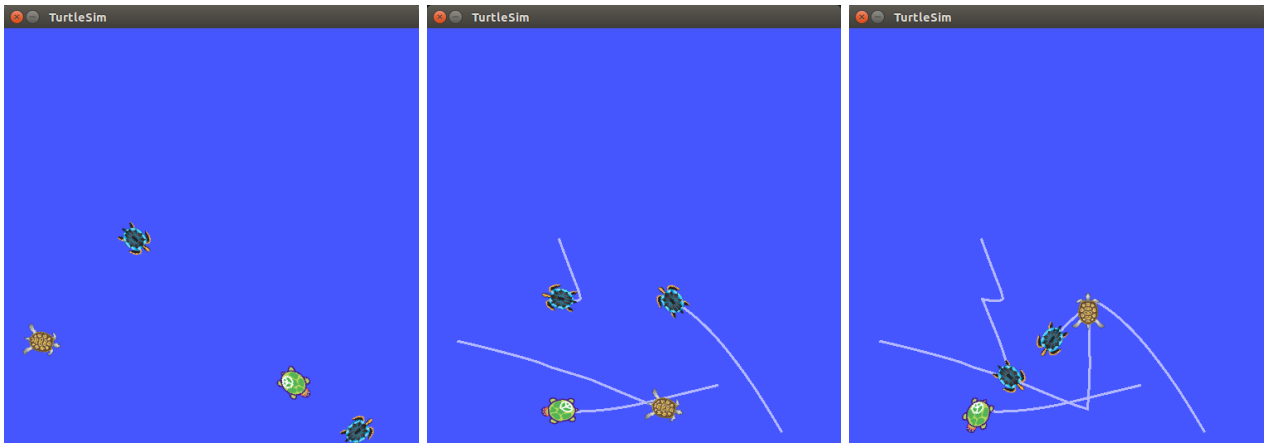
4.4 Test Results

4.4.1 Test 1

The first series of tests is conducted as follow:

1. Four turtles are spawned at random positions.
2. $$Z_1 = \begin{bmatrix} 0 & 0 \\ 0 & 3 \\ 3 & 3 \\ 3 & 0 \end{bmatrix}$$
   is given as the configuration, The turtles should converge into a square formation.
3. $$Z_2 = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}$$
   is given as the configuration. The turtles should converge into a line formation.

The following pictures display the result of each step. The turtles converged to a square formation according to the Z_1 and then converged into a line formation when the given configuration changed to Z_2.



4.4.2 Test 2

The second series of tests is conducted as follow:

1. Four turtles are spawned at random positions.
2. $$Z_1 = \begin{bmatrix} 0 & 0 \\ 0 & 3 \\ 3 & 3 \\ 3 & 0 \end{bmatrix}$$
   is given as the configuration. The turtles should converge into a square formation.
3. The turtle on the top-left corner of the formation is then manually driven away and the remaining turtles should adjust to keep the formation.

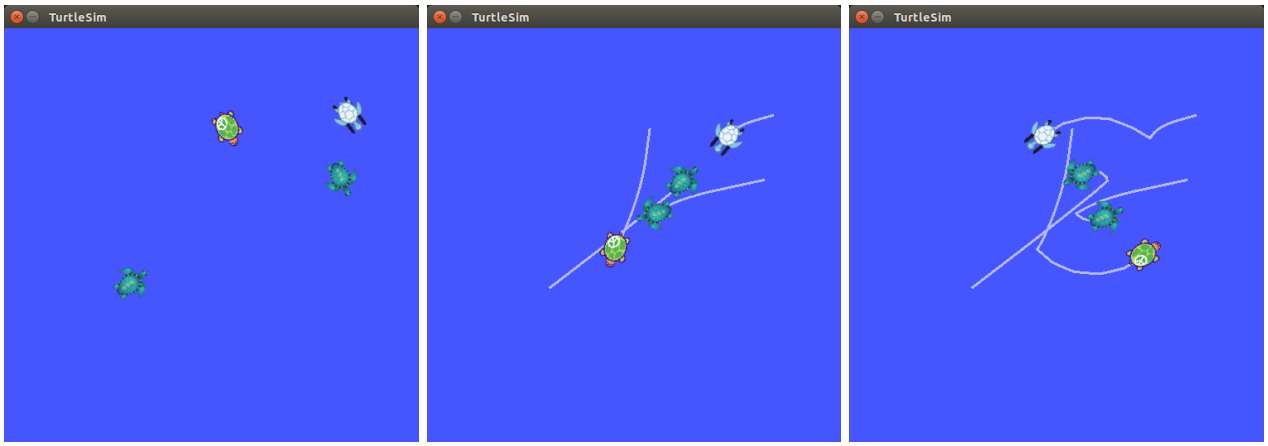
The following pictures display the result of each step and shows that the turtles converged to a square formation according to the Z_1 and was able to maintain the formation in translation.

4.4.3 Test 3

The third series of tests is conducted as follow:

1. Four turtles are spawned at random positions.
2. $$Z_2 = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}$$
   is given as the configuration. The turtles should converge into a line formation.
3. The formation is rotated 90 deg counter-clockwise and the turtles should maintain the formation.

The following pictures display the result of each step and shows that the turtles converged to a line formation according to the Z_2 and was able to maintain the formation after rotation.



With the results of the test above, we conclude that the algorithm is consistent in stabilizing the turtles into the given formation. When the formation configuration changes, the turtles regroups into the new formation. When the formation is translated or rotated, the turtles are able to maintain the formation.

5 Tips

5.1 For Formation Stabilization Algorithm

A solid understanding of paper "Global and robust formation-shape stabilization of relative sensing networks"[1] by Prof. Cortés is very helpful. To understanding the paper, I highly recommend some background in graph theory and matrix operation. Digraph[7], Laplacian matrix[8], and Kronecker product[9] are especially helpful in understanding the b-matrix2 in the update algorithm we used for the implementation.

5.2 For ROS

Going through the beginner's tutorials[10] for ROS will help with understanding the structure of ROS. The particular tutorials that should be attempt before modifying this project are #1-8, 12, 13, 15, 16 int the beginner's tutorials section. Understanding ROS rqt_graph (covered in tutorial #8) can help with understanding the graphs in section 4.3. The TurtleSim Tutorial[5] will help with understanding how Turtlesim simulation works.

6 Pitfalls

The algorithm does a great job at stabilizing the agents into the desired formation, but there is the chance for collision when transitioning from an initial state into the formation. It is not an issue with simulation conducted in TurtleSim, but for testing on the TurtleBots physically, we would need to address the collision issue during transition.

7 Conclusion

With the formation stabilization algorithm, a network of agents can maintain a formation during their task. The algorithm was implemented in python and tested using ROS framework and was found to work consistently in the 2D space. The agents would converge to the desired formation from their initial position, and the formation is maintained in translating and rotating the formation. With the exception of the concern for collision during transitioning into a formation, the algorithm is ready to be tested on TurtleBots physically.

References

[1] Jorge Cortés *Global and robust formation-shape stabilization of relative sensing networks.* Automatica 45 (12) (2009), 2754-2762 `http://carmenere.ucsd.edu/jorge/publications/data/2008_Co-auto.pdf`

[2] *Multi-Agent Robotics Lab* `http://muro.ucsd.edu/`

[3] *TurtleBot* `http://www.turtlebot.com/`

[4] *Ros* `http://www.ros.org/`

[5] *Turtlesim* `http://wiki.ros.org/turtlesim`

[6] *Rotation matrix* `https://en.wikipedia.org/wiki/Rotation_matrix`

[7] *Directed Graph* `https://en.wikipedia.org/wiki/Directed_graph`

[8] *Laplacian matrix* `https://en.wikipedia.org/wiki/Laplacian_matrix`

[9] *Kronecker product* `https://en.wikipedia.org/wiki/Kronecker_product`

[10] *Ros Tutorials* `http://wiki.ros.org/ROS/Tutorials`