Development of Leader Following, Boids Inspired Algorithm Using Robot Operating System (ROS)

Sami Alperen Akgun
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
saakgun@uwaterloo.ca

***Abstract*—In this paper, boids inspired leader following multi-robot system was implemented in Stage simulator using Robot Operating System (ROS). An additional migration part was added to three main boids rules which are separation, cohesion and alignment. Quantitative metrics were developed to calculate multi-robot system's performance. Experiments were conducted in simulation to test the proposed system and provide insight about multi-robot system design.**

***Index Terms*—BOIDS, Reynolds Rules, Multi-Robot Systems, Leader Following, ROS**

I. Introduction

Multi-robot systems have grown in popularity recently as they offer parallel execution of a task which often leads to more efficient solutions than single robot systems [1]. Moreover, one robot equipped with multiple advanced functions might be expensive to produce while single-functioning robots are often lower cost and easier to maintain [2].

As the number of robots in a system increases, the coordination of multi-robot systems becomes more difficult. The most easiest and straightforward way to solve this problem is by assigning different roles to robots in the group such as leaders and followers [3]. Depending on the application and number of the robots, the group can be divided into subsets and each subset might have a different leader; other solutions might have only one leader and the rest of the group can be followers [4]. Furthermore, researchers often take inspiration from biological systems in order to control and coordinate multi-robot systems [5]. While many animals behave independently as an individual within a group, animal groups move as if there is a central planner which coordinates individuals in the group [5]. Researcher Craig Reynolds was the first to implement a set of mathematical rules to model this kind of behaviour [6]. By programming each individual in the group (called boids) independently, he managed to obtain very natural movement of bird flocks.

After successful introduction of Reynolds rules, researchers have started to use these rules to control robot swarms. Hauert et al. used boids to create a flock of 10 drones both in simulation and real world [7], but drones didn't have separation capability of boids since they were flying in different attitudes. Braga et al. implemented a boids inspired algorithm for multi-rotor UAVs using ROS and tested in both simulation and reality [8]. However, they didn't analyze the leader-follower aspect of it. On the other hand, Carpin and Parker proposed a leader-follower algorithm for a collaborative multi-robot system [9]; yet, they used a behaviour based approach to coordinate a multi-robot system instead of Reynolds rules. On the other hand, Dunk and Abbass investigated three main Boids rules using evolutionary computation methods to include leader-follower behaviour [10]. Nonetheless, they didn't use ROS to implement their algorithm, which makes it difficult to employ in other robotic platforms. Lastly, Barisic and Krizmancic implemented three basic Boids rules using ROS and Stage simulator, but they didn't have any method to quantify the success of the algorithm [26]. They also didn't include leader following behaviour.

In this paper, implementation of Reynolds rules using ROS and Stage simulator environment was introduced. Leader following behaviour and quantitative measurement methods were added to implementation provided in [26]. Quantitative metrics related to leader following behaviour in this paper were a simplified version of the work in [10]. Nonetheless, different from them, focus of this project was on implementation. The main motivation behind this work is to provide researchers a multi-robot system which is capable of navigation and following a leader naturally in ROS environment. This system later can be used for person-following applications for multi-robot systems like [11]. Overview paper [12] can provide more information about person-following systems for interested readers.

The rest of the paper is organized as follows. In section II, the methodology is explained in a detailed manner. Section III explains experiments followed by discussion in Section IV. Finally, conclusion is made in Section V followed by limitations and future work in Section VI.

II. Methodology

A. Reynolds Rules

There are three main boids rules: separation, alignment and cohesion [6] as shown in Fig. 1.

- Separation: Robots move away from each other to avoid collisions.
- Alignment: Robots match their heading with other robots to move like group.
- Cohesion: Robots stay close to each other to form a group.

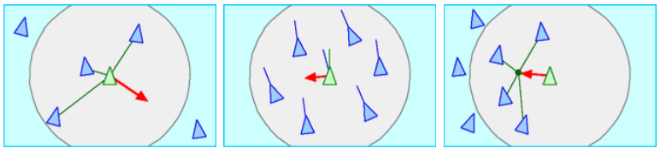

Fig. 1. Reynolds Three Main Rules: Separation (Left), Alignment (Center) and Cohesion (Right) [15]

The three behaviours and leader following behaviour was first implemented in Python programming language according to the simple algorithm structures proposed in [13]. In this way, desired boids algorithm first tested before going into details related to ROS and Stage simulator. This kind of approach makes debugging process pretty easier. The Python algorithm is published as an open source package on Github[1]. Related code can also be found in Appendix B. An example output of this implementation can be seen in Figure 2.

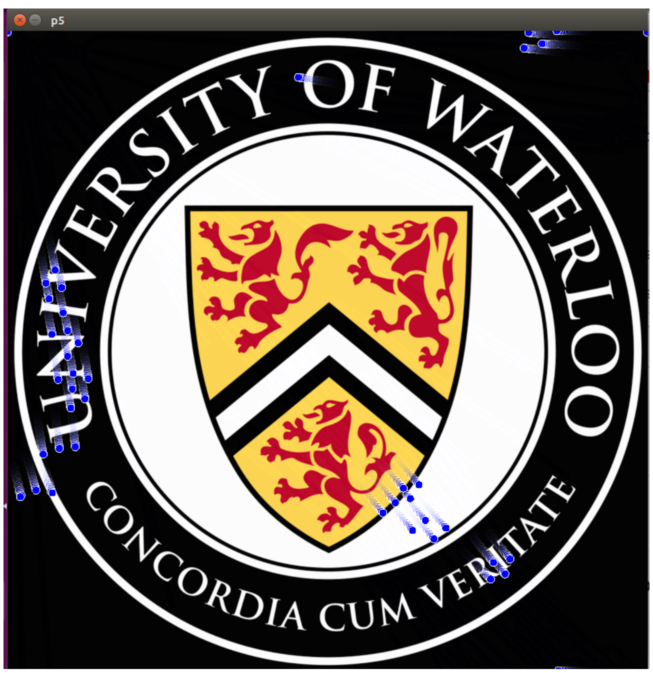

Fig. 2. Boids Python implementation (blue circles are agents)

After successful Python implementation of boids, same algorithm was implemented in ROS environment. For this purpose, I started with ROS package "sphero-formation" provided in [26]. This package was extended with leader following behaviour and quantitative metrics to measure success of overall algorithm. Other missing details were added, and unrelated scripts were removed. A new ROS package "boids_ros"[2] was created. This is the main repository used in this project. Related code regarding this package can be found in Appendix C. Overall design of the package and ROS architecture was described in the following sections.

[1]https://github.com/samialperen/boids-python
[2]https://github.com/samialperen/boids_ros

TABLE I
MAIN COMPUTER PROPERTIES

GPU	Intel HD Graphics (Coffeelake 3x8 GT2)
CPU	Intel i7-8700 CPU @ 3.20GHz × 12
OS	Ubuntu 16.04, 64 Bit
RAM	16GB DDR4

Group leader was independent from the rest of the group, so it was not included in calculations of three main boids rules. It moves independently in a specific pre-defined trajectory in order to obtain comparable results for the different cases of experiment. Rest of the group follows its trajectory thanks to additional leader following behaviour.

B. *Robots and Simulation*

In the proposal of the project, using Turtlebot3 robot had been suggested since it is highly compatible with all distributions of ROS like Indigo, Kinetic or Melodic. Yet, it wasn't preferred in the paper since I realized that starting with simple robots is more wisely considering computational power. In addition, Gazebo simulation was considered as a simulation environment, but after struggling with Gazebo for a long time, I soon recognized that it is not a good simulation environment for multi-robot systems. The reason behind this finding was instantiating multiple robots in Gazebo in a consistent, scalable way is surprisingly challenging due to dynamic transform frame calculations as highlighted by Brian Bingham, an Associate Professor in the Department of Mechanical and Aerospace Engineering at the Naval Postgraduate School in California [16], [17], [18].

Stage [19], V-REP [20], MORSE [21], Webots [22], USARSim [23] and Unity [24] simulators were considered as an alternative to Gazebo. Main properties of these simulators were presented in Table II. Although all the simulations except Stage provides 3D simulations and state of art rendering engines, Stage was used in this study since the focus of the project is on 2D simulation, and it requires less computational power than others. The main computer used in this project has the following properties stated in Table I.

As mentioned before three main rules of boids algorithm with ROS and Stage simulator was implemented in [26]. They used Sphero SPRK+ robots (Figure 3) in their project. Since Spheros are simple sphere shaped robots and my focus is on 2D simulation, it was determined to use Sphero SPRK+ robots in this project too. Nonetheless, thanks to ROS, switching robots and moving to real world applications is going to be quite straight-forward. Specific simulation environment used in this project and ROS visualization tool (Rviz) output can be seen in Figure 4 and Figure 5.

C. *ROS Architecture*

All ROS nodes and scripts was explained in this section in order to give reader a clear understanding of the implementation.

TABLE II
CHARACTERISTICS OF DIFFERENT ROBOTICS SIMULATORS [25] (AMOUNT OF * SHOWS THE LEVEL OF SUPPORT/COMPATIBILITY)

	V-REP	**Gazebo**	**MORSE**	**Webots**	**USARSim**	**Stage**	**Unity**
Main Progr. Language	C++	C++	Python	C++	C++	C++	C++
Operating System	Mac, Linux	Mac, Linux	BSD, Mac, Linux	Linux, Mac	Linux	Linux	Linux
Simulation Type	3D	3D	3D	3D	3D	2D	3D
Physics Engine	ODE, Bullet, Vortex, Newton	ODE, Bullet, Dart	Bullet	ODE	Unreal	OpenGL	Unity 3D
3D Rendering Engine	Internal, External	OGRE	Blender game	OGRE	Karma	-	OGRE
Portability	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Support	****	*****	****	****	***	****	**
ROS Compatibility	****	*****	****	***	**	****	*

Fig. 3. Sphero SPRK+ Robot [29]

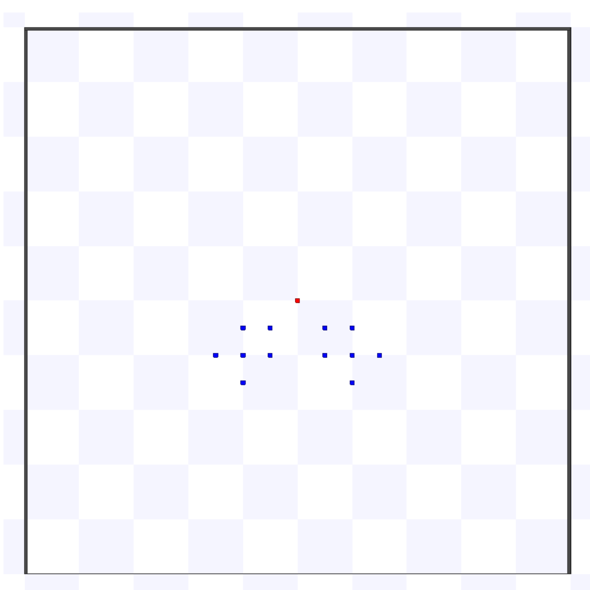

Fig. 4. Simulation environment used to evaluate boids algorithm; leader robot(red) and followers (blue)

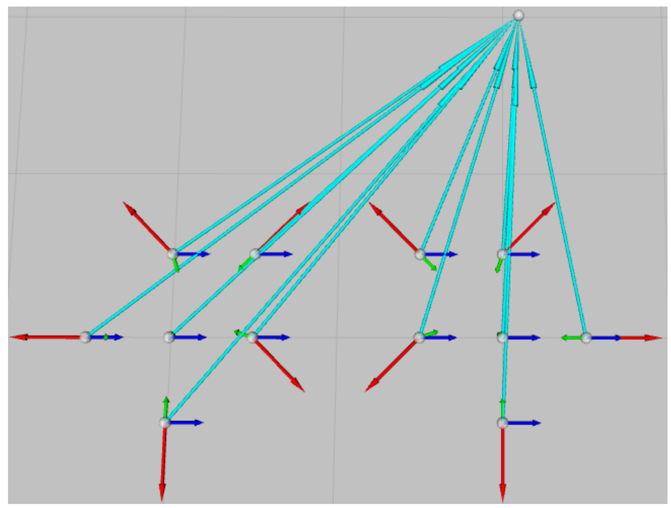

Fig. 5. Rviz output with marker arrays (blue:alignment, green:cohesion, red:separation and light blue:leader following), (leader is at top right)

When the simulation is started, ROS nodes and topics seen in Figure 6 appears. The node “simulator” publishes each robot’s odometry information which contains both their pose and velocity. This information is used with “tf (transform frame)” to dynamically update robot’s location. Then, this update is sent to “map_server” which then updates “map” topic. “dyn_reconf” node allows user to change algorithm’s parameters dynamically in real time as it can be seen in Figure 7.

After boids algorithm is started, ROS graph becomes like in Figure 8. The node “search” is responsible to calculate relative pose of neighbor agents for each member of flock including leader. This information is taken as an input by boids main node “reynolds_controller”. This node sends calculated velocity to each agent in the group using “cmd_vel” message type (excluding leader). Markers are used for display purposes in Rviz as mentioned before (see Figure 5). “Leader_controller” node specifies pre-defined trajectory of leader and guarantees that leader follows that trajectory using a simple feedback loop. As an alternative to leader controller, optional ROS node “boids_teleop” is developed to allow users to move the leader via keyboard. In the end, “rosbag_record” logs all related data in a proper ROS format for post analysis.

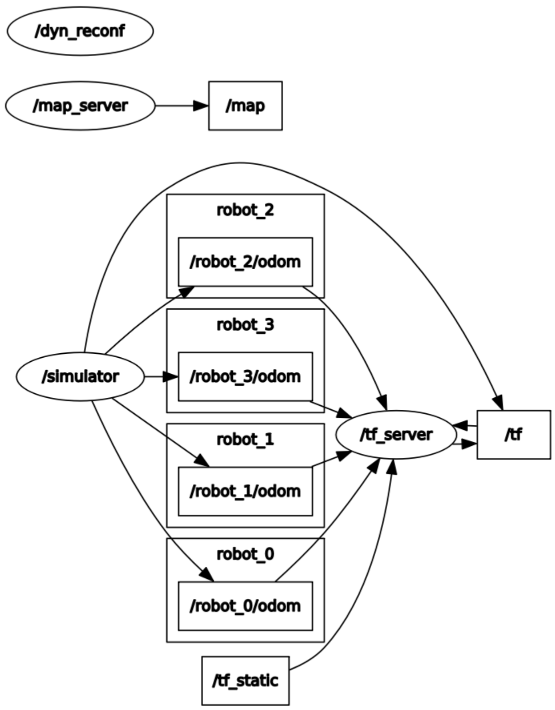

Fig. 6. ROS nodes and topics when simulation is started

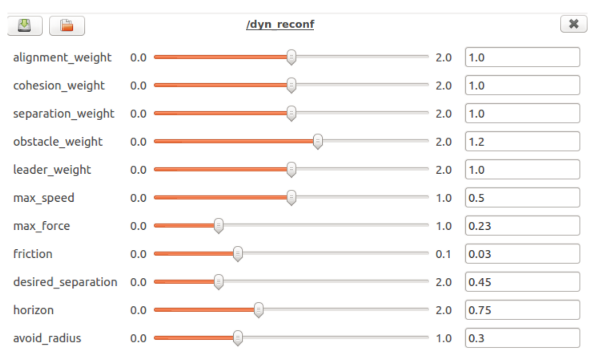

Fig. 7. Dynamic reconfiguration node to change parameters in real time

After all the data had been obtained, "data_analyzer" was called to calculate necessary metrics. Finally, "visualize_data" is responsible to display plots and make statistical analysis seen in Results section. More information related to ROS architecture and scripts can be found in Appendix C.

III. Experiments

A. Research Questions

Using the methods described in the previous section, the flocking behaviour of leader and followers was analyzed by conducting an experiment in Stage simulator. It was expected to gain more insight about the following research questions:

Research Question 1: How does velocity of leader affect boids performance?

Research Question 2: What is the relationship between relative boids parameters and boids performance?

Research Question 3: How does number of agents in the group affect boids performance?

B. Experimental Procedure

All the experiments were carried on Stage simulator as described in the previous sections. Overall view of different cases and (in)dependent variables related to different research questions can be seen in Table III. Besides, a list of of controlled variables can be seen in Table IV. To address the first research question, different leader velocity levels were compared: the same as followers' velocity, 10% more than followers' velocity and 20% more than followers' velocity like in Table III.

TABLE III
Experiment Cases, Dependent and Independent Variables

Research Q	Case ID	Independent Variable	Dependent Variable
1	1	leader velocity = 0.5m/s	violation of metrics
1	2	leader velocity = 0.55m/s	violation of metrics
1	3	leader velocity = 0.6m/s	violation of metrics
2	4	leader_weight = 1.0	violation of metrics
2	5	leader_weight = 1.1	violation of metrics
2	6	leader_weight = 1.2	violation of metrics
3	7	group size = 7	total violation amount of boids rules
3	8	group size = 13	total violation amount of boids rules
3	9	group size = 19	total violation amount of boids rules

TABLE IV
Controlled Variables

Variable Name	Description	Value
separation_weight	gain of separation	1.0
cohesion_weight	gain of cohesion	1.0
alignment_weight	gain of alignment	1.0
horizon	radius of each agent's detection range	1.0
desired_separation	separation threshold	0.7m
desired_cohesion	cohesion threshold	2.25m
desired_alignment	alignment threshold	45°
agent_speed	speed of members	0.5m/s

Regarding the second research question, leader weight was varied while keeping other weights constant as it can be seen in Table III. One can see pseudo code of main boids algorithm below to understand the logic behind the second part of the experiment.

```
def main_boids()
Vector v1, v2, v3, v4
Boid b
FOR EACH BOID b
    v1 = separation(b) * separation_weight
    v2 = alignment(b) * alignment_weight
    v3 = cohesion(b) * cohesion_weight
    v4 = leader_follow(b) * leader_weight
    b.velocity += v1 + v2 + v3 + v4
    b.position = b.position + b.velocity
end
```

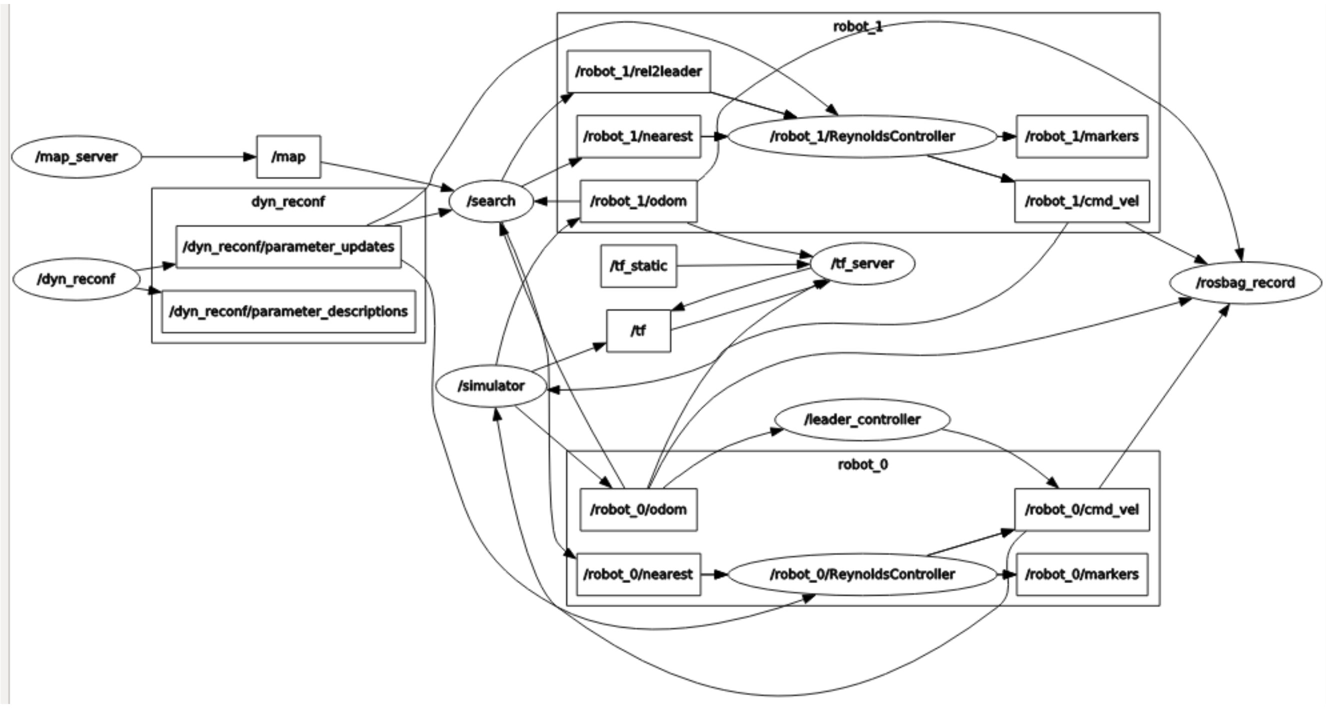

Fig. 8. ROS nodes and topics after boids algorithm is started

To address last research question, same experimental scenario with different total number of agents were examined. Due to limitations regarding computational power, experiments were conducted with either total of 7 agents, 13 agents or 19 agents. Leader was included in the total number of agents. From implementation perspective, there isn't any limit for total number of agents in the flock, but one has to provide necessary computational power and simulation area for larger multi-robot systems. For instance, for the specific computer whose properties were listed in Table I, it was discovered that total agent limit is around 30.

C. Measures

In order to analyze results, some quantitative metrics were introduced. There are different metrics for each of the Reynolds rules, so in the end total of three different metrics were proposed. All metrics were measured with respect to leader like in [10].

1) Separation: Each followers' distance to the leader was calculated. If any of the followers' distance (d_{sep}) is closer than a predefined distance, then the duration of this time interval (t_{sep}) was logged. Let's call the number of robots that violate this distance as n_{sep} and total simulation time as t_{sim}. Then, the violation of separation (V_{sep}) can be calculated:

$$V_{sep} = \frac{t_{sep} * n_{sep}}{t_{sim}} \tag{1}$$

2) Alignment: Each followers' heading orientation with respect to the leader was calculated. If any of the followers' heading orientation is more than a predefined threshold angle (θ_{align}), then the duration of this time interval (t_{align}) was stored with the number of followers that violate the alignment (n_{align}). Then, computation for violation of alignment (V_{align}) can be made:

$$V_{align} = \frac{t_{align} * n_{align}}{t_{sim}} \tag{2}$$

3) Cohesion: Each followers' distance to the leader was calculated. If any of the followers' distance (d_{coh}) is more than a predefined distance, then the duration of this time interval (t_{coh}) was logged. Let's call the number of robots that violate this distance as n_{coh} and total simulation time as t_{sim}. Then, the violation of cohesion (V_{coh}) can be calculated:

$$V_{coh} = \frac{t_{coh} * n_{coh}}{t_{sim}} \tag{3}$$

Above metrics were calculated for different cases related to research questions 1 and 2. Having larger values for these metrics means that boids performance is not high, i.e. low values of proposed metrics indicates high group performance. On the other hand, these metrics weren't calculated for cases related to last research question since it is obvious that these metrics will be accumulated as number of agents in the group increases. Therefore, violation amount of each boids rule was calculated in a discrete way for each agent. In other words, time instants in which members of the group violates boids rules were collected. Only first 6 agent's collected data was summed up as a total violation amount for each rule since the smallest population size is 6 (except leader).

D. Results

The described system was run 45 times in total and obtained results were stored in rosbag files with proper naming conventions to allow further analysis. For statistical purposes, simulation was run 5 times for each case in the experiment. One way ANOVA test [27] was used to check whether there is a statistical difference between three subgroups related to each research question. After finding a significant difference, post-hoc analysis was performed using Tukey's Honestly Significant Difference (HSD) test [28]. This has to be done since ANOVA doesn't tell which subgroups are significantly different from each other. Significance level for the tests , i.e. the threshold to reject null hypothesis, was chosen as 0.05. An example video which shows the system output during data collection can be seen in the footnote[3].

1) Variation of Leader Velocity: The results in Figure 9 was obtained when leader velocity was varied. Significant difference was calculated for violation metrics related to all boids rules. Violation of separation for the leader with 0.5 m/s velocity is significantly higher than the leader with 0.55 m/s and 0.6 m/s (p=0.001). Furthermore, violation of cohesion for the leader with 0.6 m/s is significantly higher than the case with leader velocity 0.5 m/s and 0.6m/s (p=0.001). Lastly, all the violation of alignment values for three cases are different than each other with p value 0.001.

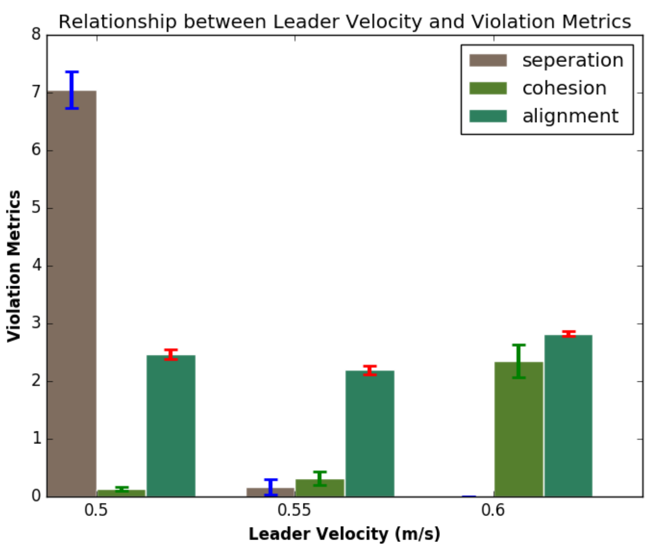

Fig. 9. Variation of Violation Metrics with Leader Velocity

2) Variation of Leader Weight: Results didn't show any significant difference in subgroups with different leader weight over other boids weights. Related results can be seen in Figure 10.

3) Variation of Number of Agents in the Group: Total separation violation during entire simulation is higher for group with 7 population than group with 13 population (p=0.001) and group with 19 population (p=0.0014). On the other hand, total

[3]https://www.youtube.com/watch?v=RTcC8k2Nvyw

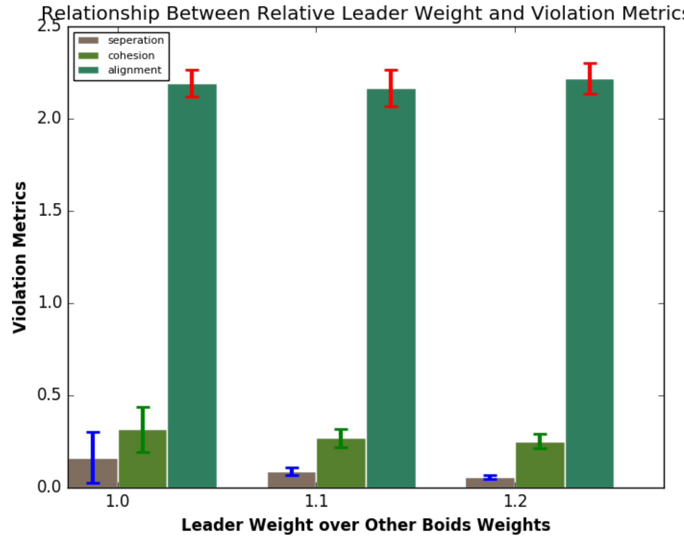

Fig. 10. Variation of Violation Metrics with Relative Leader Weight

cohesion violation during entire simulation is higher for group with 19 population than group with 7 population (p=0.001) and group with 13 population (p=0.0101). Lastly, total alignment violation during entire simulation for all populations are different than each other with p value 0.001. Results related to this case were shown in Figure 11.

Due to space limitations, all results obtained from statistical tests were not explained here, but they were attached as an appendix in the end. One can see them in Appendix A.

IV. Discussion

Overall, the following discussions can be made to interpret the obtained results in light of proposed research questions. **Research question 1** tries to find out the relation between leader velocity and boids performance. In this regard, a trade-off was discovered in the results shown in Figure 9. When leader moves as the same speed with the rest of the group, violation of separation increases, but violation of cohesion decreases. On the other hand, if leader goes so fast relative to rest of the group, although violation of separation decreases, violation of cohesion increases. This is quite natural because rest of the group can not catch the leader. Therefore, there is a trade off between violation of separation and violation of cohesion. One can not get the perfect metrics for both separation and cohesion at the same time. Choosing leader velocity around 10% more than rest of group's velocity seems a wise choice to balance violation of separation and violation of cohesion. This may be also useful for person following research in HRI since people tend to prefer robots that move slower than themselves [30].

Second research question tries to examine the effect of leader weight over other boid weights. However, findings of the experiment didn't show any difference as it can be seen in Figure 10. The reason behind this finding might cause from the selection of weights. Even 20% difference in relative leader weight may not be enough to produce any distinction, or it

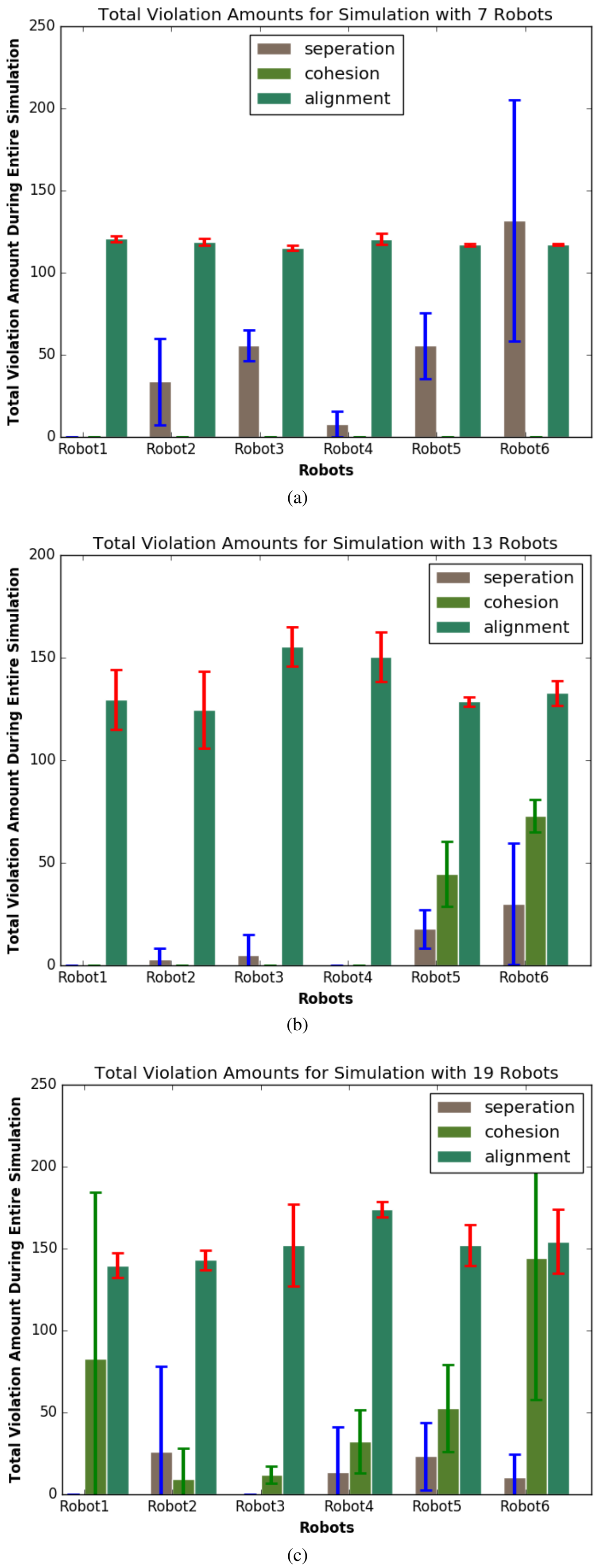

Fig. 11. Variation of Total Violation Number with Group Population

can be necessary to choose separation, cohesion and alignment weights numerically less than 1.0 (current selected value) since they can still be summed up to beat the force caused from leader weight.

Third research question is to analyze the effect of population size on boids performance. Related results can be seen in Figure 11. When multi-robot system is composed of less agents, total separation violation amount is higher. In contrast, when number of group member is increased, cohesion violation amount becomes larger in spite of reduction in separation violation amount. Hence, it can be claimed that there is a trade-off between separation violation amount and cohesion violation amount. This should be kept in mind during the design of multi-robot systems. For example, having a huge group population to reduce the task completion time might actually cause an undesirable violation amount in cohesion, i.e. multi-robots may not form a real group since all agents tries to move away from each other due to forces produced by the interaction between them and rest of the group.

On the whole, violation of alignment in Figure 9, 10 and total alignment violation amount in Figure 11 is higher than the ones for separation and cohesion. In addition, significant differences were found between each group related to each research question regarding alignment. However, all the obtained results related to alignment isn't actually meaningful. The reason behind this is the simulation setup. Simulation environment is 2D and agents are omnidirectional, so they can move to any direction without restriction. This is a good feature to allow multi-robot system to move naturally as stated in [6]. Nonetheless, this causes abrupt changes in forces that controls alignment, and it becomes difficult to measure this fashion in a discrete manner and calculate metrics.

The final point of discussion is related to variance of obtained results. Variance of results were in general high, and they increase as population size of the group increases. This was caused from the fact that simulation is stochastic and combinations of all forces between agents can cause different situations at each time simulation is run. Although this sounds negative, in reality it might be useful since real life scenarios are not deterministic as well.

V. Conclusion

In this paper, boids inspired leader following multi-robot system was implemented using Robot Operating System (ROS). Quantitative metrics to measure group performance in multi-robot systems were introduced. Experiments were conducted in Stage simulator to provide insight about system parameters and their relation to boids performance. Trade-off trend was found between varying leader velocity and separation/cohesion violation. The relation between number of agents in the group and separation/cohesion violation also had the same kind of trade-off trend.

Developed ROS architecture can provide researchers a good basis to work on multi-robot systems. Thanks to modularity and generalizability of ROS, researchers will be able to integrate their algorithms easily. Measuring Reynolds flocking rules quantitatively provides a feasible way to coordinate multi-robot groups. The decentralized nature of Reynolds rules makes adding/removing robots to already existing system quite effortless.

VI. Limitations & Future Work

First of all, system is only limited in simulation. In the future, it will be implemented in the real world by considering insights gained during this study. Secondly, the obtained results are highly depend on initial configuration of members in the group. As a future work, it would be important to analyze the effect of initial placements of group members on boids performance. In addition, pre-defined trajectory that leader followed during the experiments has compelling effect on the group performance. Thus, it would be interesting to examine consequences of different trajectories on boids performance. Lastly, selected robot type (shape, mass, dynamics) might have affected the results of the project, so it would be nice to analyze effect of robot type on the results as a future work.

During real world scenarios, leader robots might be replaced with a human so that this system can be a perfect platform to study person following behaviours or social navigation for multi-robot systems from HRI perspective. In this regard, it has a huge potential to bridge the gap between theory and practice of multi-robot systems. It is planned to use this implementation for proxemics research in multi-robot systems in the future.

Acknowledgment

I acknowledge Marko Križmančić's indirect contribution to this project for providing well written and documented open source software repository.[4]

References

[1] Modi, P.J., Shen, W.M., Tambe, M., & Yokoo, M. (2005). ADOPT: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence, 161(1), 149–180.

[2] Rubenstein, M., Ahler, C., & Nagpal, R. (2012). Kilobot: A low cost scalable robot system for collective behaviors. In IEEE International Conference on Robotics and Automation (pp. 3293–3298). IEEE.

[3] Loria, A.; Dasdemir, J.; Jarquin, N.A. Leader-follower formation and tracking control of mobile robots along straight paths. IEEE Trans. Control Syst. Technol. 2016, 24, 727–732.

[4] Consolini, L.; Morbidi, F.; Prattichizzo, D.; Tosques, M. Leader-follower formation control of nonholonomic mobile robots with input constraints. Automatica 2008, 44, 1343–1349.

[5] Moeslinger, C., Schmickl, T., Crailsheim, K. A minimalist flocking algorithm for swarm robots, European Conference on Artificial Life: 375-382, 2009.

[6] Reynolds, C. Flocks, herds and schools: A distributed behavioral model, Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques: 25-34, 1987.

[7] Hauert, S., Leven, S., Varga, M., Ruini, F., Cangelosi, A., Zufferey, J.C., Floreano, D. Reynolds flocking in reality with fixed-wing robots: communication range vs. maximum turning rate, IEEE/RSJ International Conference on Intelligent Robots and Systems: 5015-5020, 2011.

[8] R. G. Braga, R. C. da Silva, A. C. Ramos, F. Mora-Camino, Collision avoidance based on Reynolds rules: A case study using quadrotors, in Information Technology-New Generations (Springer, 2018), pp. 773–780.

[9] S. Carpin and L. E. Parker, "Cooperative leader following in a distributed multi-robot system," Proceedings of IEEE International Conference on Robotics and Automation, 2002.

[10] I. Dunk and H. Abbass, "Emergence of order in leader-follower boids-inspired systems," in Computational Intelligence (SSCI), 2016 IEEE Symposium Series on. IEEE, 2016, pp. 1–8.

[11] Shkurti F, Chang WD, Henderson P, Islam MJ, Higuera JCG, Li J, Manderson T, Xu A, Dudek G and Sattar J (2017) Underwater Multi-Robot Convoying using Visual Tracking by Detection. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE.

[12] Islam, M. J., Hong, J., & Sattar, J. (2018b). Person following by autonomous robots: A categorical overview. https://arxiv.org/abs/1803.08202

[13] Parker, C. (2007). Boids Pseudocode. [online] Kfish.org. Available at: http://www.kfish.org/boids/pseudocode.html [Accessed 14 Oct. 2019].

[14] Wiki.ros.org. (2019). rosbag - ROS Wiki. [online] Available at: http://wiki.ros.org/rosbag [Accessed 14 Oct. 2019].

[15] Red3d.com. (n.d.). Boids, Background and Update. [online] Available at: https://www.red3d.com/cwr/boids/ [Accessed 16 Nov. 2019].

[16] M. Fahad, Y. Guo, and B. Bingham, "Simulating fine-scale marine pollution plumes for autonomous robotic environmental monitoring," Frontiers Robotics AI, vol. 5, no. MAY, pp. 1–14, 2018.

[17] Bingham, B. (2019). Multi Husky Robot Simulation Wiki. [online] Available at: https://github.com/bsb808/nre_simmultihusky/wiki [Accessed 17 Nov. 2019].

[18] Bogdon, C. (2019). Simulating Multiple Husky UGVs in Gazebo - Clearpath Robotics. [online] Clearpath Robotics. Available at: https://clearpathrobotics.com/blog/2016/03/simulating-multiple-husky-ugvs-in-gazebo/ [Accessed 17 Nov. 2019].

[19] Player/Stage project. (2016) The Player Project. [Online]. Available: http://playerstage.sourceforge.net/

[20] E. Rohmer, S. P. Singh, and M. Freese, "V-rep: A versatile and scalable robot simulation framework," in Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. IEEE, 2013, pp. 1321–1326.

[21] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan. "Modular open robots simulation engine: MORSE". In Proceedings of IEEE International Conference on Robotics and Automation, pp. 46-51, 2011.

[22] O. Michel. "Webots: Symbiosis between virtual and real mobile robots". In Proceedings of International Conference on Virtual Worlds, pp. 254-263, 1998.

[23] S. Carpin, M. Lewis, J. Wang, S. Balakirsky and C. Scrapper. "USARSim: a robot simulator for research and education". In Proceedings of IEEE International Conference on Robotics and Automation, pp. 1400-1405, 2007.

[24] Y. Hu, and W. Meng. "ROSUnitySim: Development and experimentation of a real-time simulator for multi-UAV local planning". Simulation, vol. 92(10), pp. 931-944, 2016.

[25] F. M. Noori, D. Portugal, R. P. Rocha, and M. S. Couceiro, "On 3D simulators for multi-robot systems in ROS: MORSE or Gazebo?," SSRR 2017 - 15th IEEE International Symposium on Safety, Security and Rescue Robotics, Conference, pp. 19–24, 2017.

[26] Barišić, A. and Križmančić, M. (2019). Decentralized formation control for a multi-agent system of autonomous spherical robots. [online] GitHub. Available at: https://github.com/mkrizmancic/sphero_formation [Accessed 17 Nov. 2019].

[27] Ostertagova, Eva & Ostertag, Oskar. (2013). Methodology and Application of One-way ANOVA. American Journal of Mechanical Engineering. 1. 256-261. 10.12691/ajme-1-7-21.

[28] Salkind, N. J. (2010). Encyclopedia of research design Thousand Oaks, CA: SAGE Publications, Inc. doi: 10.4135/9781412961288

[29] Harveynorman.com.au. (2019). [online] Available at: https://www.harveynorman.com.au/sphero-sprk-edition-droid.html [Accessed 27 Nov. 2019].

[30] J.T. Butler and A. Agah, "Psychological Effects of Behavior Patterns of a Mobile Personal Robot," Autonomous Robots, vol. 10, 2001, pp. 185-202.

[4] https://github.com/mkrizmancic/sphero_formation

Appendix A
Further Results

Statistical tables obtained by Tukey's HSD test during the experiment can be found below.

A. Tables Related to First Research Question

Multiple Comparison of Means – Tukey HSD, FWER=0.05

group1	group2	meandiff	p–adj	lower	upper	reject
led_055_sep	led_05_sep	6.8908	0.001	6.5176	7.2639	True
led_055_sep	led_06_sep	–0.1638	0.4928	–0.537	0.2093	False
led_05_sep	led_06_sep	–7.0546	0.001	–7.4278	–6.6815	True

Multiple Comparison of Means – Tukey HSD, FWER=0.05

group1	group2	meandiff	p–adj	lower	upper	reject
led_055_coh	led_05_coh	–0.181	0.3591	–0.5192	0.1572	False
led_055_coh	led_06_coh	2.0337	0.001	1.6954	2.3719	True
led_05_coh	led_06_coh	2.2147	0.001	1.8765	2.5529	True

Multiple Comparison of Means – Tukey HSD, FWER=0.05

group1	group2	meandiff	p–adj	lower	upper	reject
led_055_align	led_05_align	0.2691	0.001	0.1376	0.4006	True
led_055_align	led_06_align	0.6283	0.001	0.4968	0.7599	True
led_05_align	led_06_align	0.3592	0.001	0.2277	0.4907	True

B. Tables Related to Second Research Question

Multiple Comparison of Means – Tukey HSD, FWER=0.05

group1	group2	meandiff	p–adj	lower	upper	reject
weight_10_sep	weight_11_sep	–0.0754	0.409	–0.227	0.0761	False
weight_10_sep	weight_12_sep	–0.1077	0.182	–0.2593	0.0438	False
weight_11_sep	weight_12_sep	–0.0323	0.8309	–0.1839	0.1193	False

Multiple Comparison of Means – Tukey HSD, FWER=0.05

group1	group2	meandiff	p–adj	lower	upper	reject
weight_10_coh	weight_11_coh	–0.0464	0.6864	–0.1961	0.1033	False
weight_10_coh	weight_12_coh	–0.0638	0.5122	–0.2135	0.0859	False
weight_11_coh	weight_12_coh	–0.0174	0.9	–0.1671	0.1323	False

Multiple Comparison of Means – Tukey HSD, FWER=0.05

group1	group2	meandiff	p–adj	lower	upper	reject
weight_10_align	weight_11_align	–0.0272	0.8966	–0.1881	0.1336	False
weight_10_align	weight_12_align	0.024	0.9	–0.1369	0.1848	False
weight_11_align	weight_12_align	0.0512	0.6736	–0.1097	0.212	False

C. Tables Related to Third Research Question

Multiple Comparison of Means – Tukey HSD, FWER=0.05

group1	group2	meandiff	p–adj	lower	upper	reject
total_13_sep	total_19_sep	3.0	0.9	−20.0899	26.0899	False
total_13_sep	total_7_sep	38.1333	0.001	15.0435	61.2232	True
total_19_sep	total_7_sep	35.1333	0.0014	12.0435	58.2232	True

Multiple Comparison of Means – Tukey HSD, FWER=0.05

group1	group2	meandiff	p–adj	lower	upper	reject
total_13_coh	total_19_coh	35.9667	0.0101	7.2656	64.6677	True
total_13_coh	total_7_coh	−19.5667	0.2406	−48.2677	9.1344	False
total_19_coh	total_7_coh	−55.5333	0.001	−84.2344	−26.8323	True

Multiple Comparison of Means – Tukey HSD, FWER=0.05

group1	group2	meandiff	p–adj	lower	upper	reject
total_13_align	total_19_align	15.6667	0.001	6.6957	24.6376	True
total_13_align	total_7_align	−18.7333	0.001	−27.7043	−9.7624	True
total_19_align	total_7_align	−34.4	0.001	−43.371	−25.429	True

APPENDIX B
BOIDS PYTHON IMPLEMENTATION

This appendix contains the code used in boids-python repository[5]. This repository has two main scripts which are "boids.py" and "main.py".

Author: Sami Alperen Akgun

A. boids.py

```
from p5 import circle, stroke, fill
import numpy as np
class Boid(object):
def __init__(self,width,height,position,horizon, max_speed, rule1W, rule2W, rule3W
            ,desired_seperation):
    # Width, height = Screen Output Dimensions
    # x,y = boids positions
    # horizon = It describes how far boid can detect the others
    # max_speed = Max speed of each individual in the group
    # Rule1 = Cohesion , Rule2 = Seperation, Rule3= Alignment
    # rule1W = Weight for the rule1 (as a percentage), i.e. rule1W = 5 --> 5%
    # desired_seperation = Minimum distance between each boid
    self.width = width
    self.height = height
    self.position = position
    self.max_speed = max_speed
    initial_random_velocity = (np.random.rand(2)-0.5) * self.max_speed * 2
    self.velocity = initial_random_velocity
    self.horizon = horizon
    self.rule1W = rule1W
    self.rule2W = rule2W
    self.rule3W = rule3W
    self.desired_seperation = desired_seperation


def show_boid(self):
    stroke(255) #white contour colors
    fill(0,0,255) #fill with blue
    circle( (self.position[0],self.position[1]) ,radius=10)

def update_boid(self):
    # Limiting the speed
    if np.linalg.norm(self.velocity) > self.max_speed:
        self.velocity = (self.velocity/np.linalg.norm(self.velocity)) * self.max_speed
    # Then update the position
    self.position = np.add(self.position, self.velocity)

def bound_position(self):
    # If boids reach the edges, it should come back from other side
    if self.position[0] > self.width-1:
        self.position[0] = 0
    elif self.position[1] > self.height-1:
        self.position[1] = 0
    elif self.position[0] < 0:
        self.position[0] = self.width-1
```

[5] https://github.com/samialperen/boids-python

```
        elif self.position[1] < 0:
            self.position[1] = self.height-1


    def main_boid(self, boids):
        v1 = self.rule1(boids)
        v2 = self.rule2(boids)
        v3 = self.rule3(boids)

        self.bound_position()
        self.show_boid()
        self.velocity += v1 + v2 + v3
        self.update_boid()

    # This function is used to move flock to a desired position
    # desired_position = Desired target position to move boids
    # step_size = determines how much boids will move towards to desired position
    # in each iteration as a percent --> step_size = 1 means 1% at each step
    def tend_to_place(self,desired_position,step_size):
        self.velocity = (desired_position - self.position) * (step_size / 100)

    def rule1(self,boids): #Cohesion
        center_of_mass = np.zeros(2)
        N = 0 #Total boid number

        for b in boids:
            # self is the boid we are currently looking for. We don't want to take its
            #  position into account for center of mass that's why we have the expression
            # right of &
            if (np.linalg.norm(b.position - self.position) < self.horizon) & (b != self):
                center_of_mass += b.position
            N += 1

        center_of_mass = center_of_mass / (N-1)
        target_position = (center_of_mass * self.rule1W) / 100

        return target_position

    def rule2(self,boids): #Seperation
        c = np.zeros(2)
        for b in boids:
            if ( (np.linalg.norm(b.position - self.position) < self.horizon)
                    & (np.linalg.norm(b.position - self.position) < self.desired_seperation)
                    & (b != self) ): #end of condition
                c -= (b.position - self.position)*(self.rule2W/100) #end of if

        return c

    def rule3(self,boids): #Alignment
        perceived_velocity = np.zeros(2)
        N = 0 #Total boid number

        for b in boids:
            if (np.linalg.norm(b.position - self.position) < self.horizon) & (b != self):
                perceived_velocity += b.velocity
            N += 1
```

```
    perceived_velocity = perceived_velocity / (N-1)
    pv = (perceived_velocity * self.rule3W) / 100

    return pv
```

B. main.py

```
from p5 import *
import numpy as np
from boids import Boid

# Parameters for visualization
bg = None
width = 800
height = 800

# Parameters regarding flocks for description look boids.py
horizon = 100
max_speed = 2
rule1W = 100
rule2W = 100
rule3W = 100
desired_seperation = 20
#desired_position = np.array([100,600]) #You can give static desired positions
desired_position = np.zeros(2,dtype=np.int32)
step_size = 10

# Create flocks
N = 40 #Total number of boids
flock = [None for _ in range(N)]
for i in range(N):
    initial_position = np.zeros(2, dtype=np.int32)
    initial_position[0] = np.random.randint(0,width-10) # x coordinate
    initial_position[1] = np.random.randint(0,height-10) # y coordinate
    flock[i] = Boid(width,height,initial_position,horizon,max_speed,rule1W,rule2W,\
                    rule3W, desired_seperation)

def setup():
    global bg
    size(width,height) #Background image is width x height
    bg = load_image("images/UW_background.png")

def draw(): #This is the main loop for p5 library
    global flock
    background(bg)

    for boid in flock:
        boid.tend_to_place(desired_position,step_size)
        boid.main_boid(flock)

# When you click the mouse on the output, desired position
# becomes the cursor position
def mouse_pressed():
    print("Desired location: %d,%d " %(mouse_x,mouse_y) )
```

```
    desired_position = np.array([mouse_x,mouse_y])

run() #This is the main function of p5 library that calls setup once and draw in loop
```

Appendix C
Boids ROS Implementation

The repository named “boids-ros”[6] was created on top of the repository “sphero-formation”[7] provided by Marko Križmančić. In this appendix, only related part of the code will be explained indicating authors and modifications.

All scripts names in boids-ros repository and their functions with authors can be found in the table V. Script names with * are the ones that I actually contributed considerably to original code. One can also see the actual code below as attached.

TABLE V
Submodules in boids-ros repository

Script Name	Authors	Explanation
boids.py*	Marko Križmančić, rewritten by Sami Alperen Akgun	main boids algorithm
boids_teleop.py	Sami Alperen Akgun	Allow users to control leader via keyboard
data_analyzer.py	Sami Alperen Akgun	Calculates performance metrics
visualize_data.py	Sami Alperen Akgun	Creates graphs using data and make statistical analysis
dynamic_reconfiguration_node.py	Marko Križmančić, modified by Sami Alperen Akgun	Allows dynamic change of boids parameters
leader_controller.py	Sami Alperen Akgun	Moves leader in a pre-defined trajectory
nearest_search.py*	Marko Križmančić, modified by Sami Alperen Akgun	Publishes ROS topics for neighbor agents
reynolds_controller.py	Marko Križmančić, modified by Sami Alperen Akgun	Controls boids movements through ROS
simulation_tf.py	Marko Križmančić	transform frame calculation across agents
util.py	Marko Križmančić, modified by Sami Alperen Akgun	contains utility and helper functions
setup_sim.launch	Marko Križmančić, modified by Sami Alperen Akgun	starts simulation with proper configuration
reynolds_sim.launch	Marko Križmančić, modified by Sami Alperen Akgun	runs boids algorithm with all nodes

CODE WRITTEN ONLY BY ME:

A. *boids_teleop.py*

```
#!/usr/bin/env python

from __future__ import print_function

import rospy, roslib

from geometry_msgs.msg import Twist

import sys, select, termios, tty

msg = """
Reading from the keyboard and Publishing to Twist!
---------------------------
Moving around:
   q w e
   a s d
   z x c

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U I O
   J K L
   M < >

```

[6]https://github.com/samialperen/boids_ros
[7]https://github.com/mkrizmancic/sphero_formation

```
t : up (+z)
b : down (-z)

anything else : stop

i/k : increase/decrease max speeds by 10%
u/j : increase/decrease only linear speed by 10%
o/l : increase/decrease only angular speed by 10%

CTRL-C to quit
"""

moveBindings = {
      'w':(1,0,0,0),
      'e':(1,0,0,-1),
      'a':(0,0,0,1),
      'd':(0,0,0,-1),
      'q':(1,0,0,1),
      'x':(-1,0,0,0),
      'c':(-1,0,0,1),
      'z':(-1,0,0,-1),
      'E':(1,-1,0,0),
      'W':(1,0,0,0),
      'A':(0,1,0,0),
      'D':(0,-1,0,0),
      'Q':(1,1,0,0),
      'X':(-1,0,0,0),
      'C':(-1,-1,0,0),
      'Z':(-1,1,0,0),
      't':(0,0,1,0),
      'b':(0,0,-1,0),
   }

speedBindings={
      'i':(1.1,1.1),
      'k':(.9,.9),
      'u':(1.1,1),
      'j':(.9,1),
      'o':(1,1.1),
      'l':(1,.9),
   }

def getKey():
   tty.setraw(sys.stdin.fileno())
   select.select([sys.stdin], [], [], 0)
   key = sys.stdin.read(1)
   termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
   return key


def vels(speed,turn):
   return "currently:\tspeed %s\tturn %s " % (speed,turn)

if __name__=="__main__":
   settings = termios.tcgetattr(sys.stdin)
```

```
    pub = rospy.Publisher('robot_0/cmd_vel', Twist, queue_size = 1)
    rospy.init_node('boids_teleop')

    speed = rospy.get_param("~speed", 0.5)
    turn = rospy.get_param("~turn", 1.0)
    x = 0
    y = 0
    z = 0
    th = 0
    status = 0

    try:
        print(msg)
        print(vels(speed,turn))
        while(1):
            key = getKey()
            if key in moveBindings.keys():
                x = moveBindings[key][0]
                y = moveBindings[key][1]
                z = moveBindings[key][2]
                th = moveBindings[key][3]
            elif key in speedBindings.keys():
                speed = speed * speedBindings[key][0]
                turn = turn * speedBindings[key][1]

                print(vels(speed,turn))
                if (status == 14):
                    print(msg)
                status = (status + 1) % 15
            else:
                x = 0
                y = 0
                z = 0
                th = 0
                if (key == '\x03'):
                    break

            twist = Twist()
            twist.linear.x = x*speed; twist.linear.y = y*speed; twist.linear.z = z*speed;
            twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = th*turn
            pub.publish(twist)

    except Exception as e:
        print(e)

    finally:
        twist = Twist()
        twist.linear.x = 0; twist.linear.y = 0; twist.linear.z = 0
        twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = 0
        pub.publish(twist)

        termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
```

B. data_analyzer.py

```
"""
```

```
This python script reads all agents odometry and velocity from bag files
and calculates necessary metrics to quantify boids algorithm.
"""

import rosbag
import rospy
import numpy as np
import pandas as pd
import sys #for parser


# This function is used to measure euclidian distance between pd dataframes
def get_distance(a, b):
   # a, b --> pandas dataframes, inputs
   # output --> difference panda dataframe which contains
   # eucledian distances for all times
   distance = pd.DataFrame(columns=['distance','t'])
   distance['t'] = a['t']
   x_dif_square = np.square(a['x']-b['x'])
   y_dif_square = np.square(a['y']-b['y'])
   distance['distance'] = np.sqrt(x_dif_square + y_dif_square)
   return distance

# This function returns |a-b|
def get_abs_difference(a, b):
   # a, b --> pandas dataframes, inputs
   # output --> |a-b|

   difference = pd.DataFrame(columns=['angle','t'])
   difference['t'] = a['t']
   difference['angle'] = abs(a['angle']-b['angle'])
   return difference


#################### Parameters #########################################
separation_threshold = 0.7 #in meters
alignment_threshold = 45.0 # degree
cohesion_threshold = 2.25 #meter
#total_num_of_robots = rospy.get_param("/num_of_robots")
total_num_of_robots = 19


#################### Read Bag File ######################################
bagname = sys.argv[1]
bag = rosbag.Bag('../bagfiles/' + bagname + '.bag') #Read bag

############## General parameters obtained from rosbag
# The data between start_time and end_time will be analyzed
start_time = bag.get_start_time()
end_time = bag.get_end_time()
total_time = end_time - start_time

topics = bag.get_type_and_topic_info()[1].keys() #All topics in rosbag

############## Read poses
# Read leader pose --> for our case only pose.x and pose.y is enough (2D)
```

```
df_leader_poses = pd.DataFrame(columns=['x','y','t'])
leader_counter = 0
for _ , msg, t in bag.read_messages("/robot_0/odom"):
      df_leader_poses.loc[leader_counter] = [msg.pose.pose.position.x, msg.pose.pose.
         ↪ position.y, t.to_sec()]
      leader_counter += 1

# Remove duplicate time instants and take last one of them as a true value
df_leader_poses.drop_duplicates(subset='t', keep = 'last', inplace = True)
df_leader_poses = df_leader_poses.reset_index(drop=True)

# Total number of pose msgs leader published, this will be used to synchorize boids
leader_pose_msg_size = df_leader_poses.shape[0]

boids_poses = {} # Dictionary for all boids poses through time
# Example: boids[1] contains all poses for robot_1
# boids[total_num_of_robots] contains all poses for robot_total_num_of_robots
for robot_idx in range(1,total_num_of_robots): #start from robot_1
   boids_poses[robot_idx] = pd.DataFrame(columns=['x','y','t'])
   row_idx = 0
   for topic , msg, t in bag.read_messages("/robot_" + str(robot_idx) + "/odom"):
         boids_poses[robot_idx].loc[row_idx] = [msg.pose.pose.position.x, msg.pose.
            ↪ pose.position.y
                                  , t.to_sec()]
         row_idx += 1


for robot_idx in range(1,total_num_of_robots):
   # Remove duplicate time instants and take last one of them as a true value
   boids_poses[robot_idx].drop_duplicates(subset='t', keep = 'last', inplace = True)
   boids_poses[robot_idx] = boids_poses[robot_idx].reset_index(drop=True)
   min_pose_robot_index = 0 # leader
   min_pose_msg_size = leader_pose_msg_size
   if boids_poses[robot_idx].shape[0] < min_pose_msg_size:
      min_pose_msg_size = boids_poses[robot_idx].shape[0]
      min_pose_robot_index = robot_idx

###### Data Check ############
#print("############### POSE DATA CHECK ###########")
#print(df_leader_poses.shape)
#print(boids_poses[1].shape)
#print(boids_poses[2].shape)
#print(boids_poses[3].shape)
#print(boids_poses[4].shape)
#print(boids_poses[5].shape)
#print(boids_poses[6].shape)
#print(boids_poses[7].shape)
#print(boids_poses[8].shape)
#print(boids_poses[9].shape)
#print(boids_poses[10].shape)
#print(boids_poses[11].shape)
#print(boids_poses[12].shape)



## This part is to make sure that all obtained data is synchorized
```

```
for robot_idx in range(1,total_num_of_robots):
    if boids_poses[robot_idx].shape[0] > min_pose_msg_size:
        d = boids_poses[robot_idx].shape[0] - min_pose_msg_size #number of rows to
            ↪ delete
        for _ in range(d):
            if min_pose_robot_index == 0: #leader has the smallest size
                if boids_poses[robot_idx]['t'][0] != df_leader_poses['t'][0]:
                    # We are deleting first row
                    boids_poses[robot_idx] = boids_poses[robot_idx].iloc[1:,].reset_index(
                        ↪ drop=True)
                else:
                    # We need to delete last row
                    boids_poses[robot_idx] = boids_poses[robot_idx][:-1]
            else: # some agent other than leader has the smallest size
                if boids_poses[robot_idx]['t'][0] != boids_poses[min_pose_robot_index]['t'
                    ↪ ][0]:
                    # We are deleting first row
                    boids_poses[robot_idx] = boids_poses[robot_idx].iloc[1:,].reset_index(
                        ↪ drop=True)
                else:
                    # We need to delete last row
                    boids_poses[robot_idx] = boids_poses[robot_idx][:-1]

if min_pose_robot_index != 0: #leader doesn't have the smallest size
    d = df_leader_poses.shape[0] - min_pose_msg_size #number of rows to delete
    for _ in range(d):
        if df_leader_poses['t'][0] != boids_poses[min_pose_robot_index]['t'][0]:
            # We are deleting first row
            df_leader_poses = df_leader_poses.iloc[1:,].reset_index(drop=True)
        else:
            # We need to delete last row
            df_leader_poses = df_leader_poses[:-1]


############## Read orientations
# Read leader orientation --> Since it is 2D, we need to subscribe cmd_vel
# arctan(cmd_vel.linear.y / cmd_vel.linear.x) will give the orientation, i.e. angle
df_leader_angles = pd.DataFrame(columns=['angle','t'])
leader_counter = 0
for _ , msg, t in bag.read_messages("/robot_0/cmd_vel"):
        df_leader_angles.loc[leader_counter] = [np.degrees(np.arctan2(msg.linear.y,msg.
            ↪ linear.x))
                                    , t.to_sec()]
        leader_counter += 1

# Remove duplicate time instants and take last one of them as a true value
df_leader_angles.drop_duplicates(subset='t', keep = 'last', inplace = True)
df_leader_angles = df_leader_angles.reset_index(drop=True)

# Total number of orientation msgs leader published, this will be used to synchorize
    ↪ boids
leader_angle_msg_size = df_leader_angles.shape[0]

boids_angles = {} # Dictionary for all boids angles through time
# Example: boids[1] contains all angles for robot_1
# boids[total_num_of_robots] contains all angles for robot_total_num_of_robots
```

```
for robot_idx in range(1,total_num_of_robots): #start from robot_1
   boids_angles[robot_idx] = pd.DataFrame(columns=['angle','t'])
   row_idx = 0
   for topic , msg, t in bag.read_messages("/robot_" + str(robot_idx) + "/cmd_vel"):
         boids_angles[robot_idx].loc[row_idx] = [np.degrees(np.arctan2(msg.linear.y,
            ↪ msg.linear.x))
                                            , t.to_sec()]
         row_idx += 1

for robot_idx in range(1,total_num_of_robots):
   # Remove duplicate time instants and take last one of them as a true value
   boids_angles[robot_idx].drop_duplicates(subset='t', keep = 'last', inplace = True)
   boids_angles[robot_idx] = boids_angles[robot_idx].reset_index(drop=True)
   min_angle_robot_index = 0 #leader
   min_angle_msg_size = leader_angle_msg_size
   if boids_angles[robot_idx].shape[0] < min_angle_msg_size:
      min_angle_msg_size = boids_angles[robot_idx].shape[0]
      min_angle_robot_index = robot_idx

###### Data Check ############
#print("################ ANGLE DATA CHECK ############")
#print(df_leader_angles.shape)
#print(boids_angles[1].shape)
#print(boids_angles[2].shape)
#print(boids_angles[3].shape)
#print(boids_angles[4].shape)
#print(boids_angles[5].shape)
#print(boids_angles[6].shape)
#print(boids_angles[7].shape)
#print(boids_angles[8].shape)
#print(boids_angles[9].shape)
#print(boids_angles[10].shape)
#print(boids_angles[11].shape)
#print(boids_angles[12].shape)


## This part is to make sure that all obtained data is synchorized
for robot_idx in range(1,total_num_of_robots):
   if boids_angles[robot_idx].shape[0] > min_angle_msg_size:
      d = boids_angles[robot_idx].shape[0] - min_angle_msg_size #number of rows to
         ↪ delete
      for _ in range(d):
         if min_angle_robot_index == 0: #leader has the smallest size
            if boids_angles[robot_idx]['t'][0] != df_leader_angles['t'][0]:
               # We are deleting first row
               boids_angles[robot_idx] = boids_angles[robot_idx].iloc[1:,].reset_index
                  ↪ (drop=True)
            else:
               # We need to delete last row
               boids_angles[robot_idx] = boids_angles[robot_idx][:-1]
         else: # some agent other than leader has the smallest size
            if boids_angles[robot_idx]['t'][0] != boids_angles[min_angle_robot_index][
               ↪ 't'][0]:
               # We are deleting first row
               boids_angles[robot_idx] = boids_angles[robot_idx].iloc[1:,].reset_index
                  ↪ (drop=True)
```

```
            else:
                # We need to delete last row
                boids_angles[robot_idx] = boids_angles[robot_idx][:-1]

if min_angle_robot_index != 0: #leader doesn't have the smallest size
   d = df_leader_angles.shape[0] - min_angle_msg_size #number of rows to delete
   for _ in range(d):
      if df_leader_angles['t'][0] != boids_angles[min_angle_robot_index]['t'][0]:
         # We are deleting first row
         df_leader_angles = df_leader_angles.iloc[1:,].reset_index(drop=True)
      else:
         # We need to delete last row
         df_leader_angles = df_leader_angles[:-1]








#################### Calculate Metrics ######################################


##### Calculate relative distance to leader for each robot for all the time
boids_rel2leader_poses = {} # Dictionary for all boids distances to leader
# E.g. boids_rel2leader[2] will contain distance of robot_2 to leader for all the
   ↪ times
# boids_rel2leader[2] structure will be an pd dataframe with columns--> distance and t
for robot_idx in range(1,total_num_of_robots):
   boids_rel2leader_poses[robot_idx] = get_distance(df_leader_poses, boids_poses[
      ↪ robot_idx])
   print("#########Robot_Poses_%d" %(robot_idx))
   print(boids_rel2leader_poses[robot_idx])

##### Calculate relative angles to leader for each robot for all the time
boids_rel2leader_angles = {} # Dictionary for all boids angles relative to leader
# E.g. boids_rel2leader[2] will contain distance of robot_2 to leader for all the
   ↪ times
# boids_rel2leader[2] structure will be an pd dataframe with columns--> distance and t
for robot_idx in range(1,total_num_of_robots):
   boids_rel2leader_angles[robot_idx] = get_abs_difference(df_leader_angles,
      ↪ boids_angles[robot_idx])
   print("#########Robot_Angles_%d" %(robot_idx))
   print(boids_rel2leader_angles[robot_idx])

##### Separation Metric
Q_sep_nominator = 0.0
for robot_idx in range(1,total_num_of_robots):
   t_sep = 0.0 #for each boid we are calculating separately
   total_sep_violation = 0 #Number of time instants one boid violates seperation
   seperation_check = boids_rel2leader_poses[robot_idx]['distance'] <
      ↪ separation_threshold
   total_sep_violation = boids_rel2leader_poses[robot_idx]['t'][seperation_check].
      ↪ shape[0]
   print("Seperation_violation:_%d" %(total_sep_violation))
```

```
    if total_sep_violation != 0:
        t_sep = total_sep_violation * 0.1 #There is 0.1 time difference between time instants
        Q_sep_nominator += t_sep

Q_sep = Q_sep_nominator / total_time #violation of seperation

##### Cohesion Metric
Q_coh_nominator = 0.0
for robot_idx in range(1,total_num_of_robots):
    t_coh = 0.0 #for each boid we are calculating separately
    total_coh_violation = 0 #number of time instants one boid violates cohesion
    cohesion_check = boids_rel2leader_poses[robot_idx]['distance'] > cohesion_threshold
    total_coh_violation = boids_rel2leader_poses[robot_idx]['t'][cohesion_check].shape[0]
    print("Cohesion violation: %d" %(total_coh_violation))
    if total_coh_violation != 0:
        t_coh = total_coh_violation * 0.1 #There is 0.1 time difference between time instants
        Q_coh_nominator += t_coh

Q_coh = Q_coh_nominator / total_time #violation of seperation


##### Alignment Metric
Q_alig_nominator = 0.0
for robot_idx in range(1,total_num_of_robots):
    t_alig = 0.0 #for each boid we are calculating separately
    total_alig_violation = 0 #number of time instants one boid violates cohesion
    alignment_check = boids_rel2leader_angles[robot_idx]['angle'] > alignment_threshold
    total_alig_violation = boids_rel2leader_angles[robot_idx]['t'][alignment_check].shape[0]
    print("Alignment violation: %d" %(total_alig_violation))
    if total_alig_violation != 0:
        t_alig = total_alig_violation * 0.1 #There is 0.1 time difference between time instants
        Q_alig_nominator += t_alig

Q_alig = Q_alig_nominator / total_time #violation of seperation


print("Q_sep: %f" %(Q_sep))
print("Q_coh: %f" %(Q_coh))
print("Q_alig: %f" %(Q_alig))




#pd.set_option('display.max_rows', 1000)
bag.close()
```

C. *visualize_data.py*

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats
```

```python
import pandas as pd
from statsmodels.stats.multicomp import (pairwise_tukeyhsd,MultiComparison)

"""
Research Question 1 --> Relation between metrics and leader speed
Controlled Variables: separation threshold = 0.7m , cohesion threshold = 2.25m,
alignment threshold = 45 degree, total number of robots = 13, delay = 1.8s
all boids weights (4 of them) are 1.0
Independent variable: leader velocity 0.5 or 0.55 or 0.6 m/s
"""
led_05_sep = [7.418773, 6.908654, 6.589588, 6.962874, 7.393245]
led_05_coh = [0.105897, 0.176683, 0.181598, 0.101796, 0.108565]
led_05_align = [2.547533, 2.451923, 2.340194, 2.402395, 2.572979]

led_055_sep = [0.174263, 0.052209, 0.395442, 0.0, 0.197315]
led_055_coh = [0.225201, 0.360107, 0.222520, 0.538255, 0.233557]
led_055_align = [2.076408, 2.228916, 2.268097, 2.249664, 2.146309]

led_06_sep = [0.0, 0.0, 0.0, 0.0, 0.0]
led_06_coh = [2.441691, 1.906841, 2.676385, 2.150146, 2.572886]
led_06_align = [2.864431, 2.799127, 2.750729, 2.857143, 2.839650]

bar_width = 0.25

# set height of bar
bars_sep = [np.mean(led_05_sep),np.mean(led_055_sep),np.mean(led_06_sep)]
bars_coh = [np.mean(led_05_coh),np.mean(led_055_coh),np.mean(led_06_coh)]
bars_align = [np.mean(led_05_align),np.mean(led_055_align),np.mean(led_06_align)]

# set standard deviation of data for error bars
sep_std = [np.std(led_05_sep), np.std(led_055_sep), np.std(led_06_sep)]
coh_std = [np.std(led_05_coh), np.std(led_055_coh), np.std(led_06_coh)]
align_std = [np.std(led_05_align), np.std(led_055_align), np.std(led_06_align)]


# Debug
print("Separation:")
print(bars_sep)
print("Cohesion:")
print(bars_coh)
print("Alignment:")
print(bars_align)

# Set position of bar on X axis
r1 = np.arange(len(bars_sep))
r2 = [x + bar_width for x in r1]
r3 = [x + bar_width for x in r2]

# Make the plot
plt.bar(r1, bars_sep, yerr=sep_std, error_kw=dict(lw=3, capsize=5, capthick=2), color=
    ↪ '#7f6d5f', width=bar_width, edgecolor='white', label='seperation')
plt.bar(r2, bars_coh, yerr=coh_std, error_kw=dict(lw=3, capsize=5, capthick=2), color=
    ↪ '#557f2d', width=bar_width, edgecolor='white', label='cohesion')
plt.bar(r3, bars_align, yerr=align_std, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#2d7f5e', width=bar_width, edgecolor='white', label='alignment')
```

```
# Add xticks on the middle of the group bars
plt.xlabel('Leader Velocity (m/s)', fontweight='bold')
plt.ylabel('Violation Metrics', fontweight='bold')
plt.xticks([r + bar_width for r in range(len(bars_sep))], ['0.5','0.55','0.6'])
plt.title('Relationship between Leader Velocity and Violation Metrics')
# Create legend & Show graphic
plt.legend()
#plt.savefig('leader_velocity.png')
plt.show()

# One way ANOVA analysis for statistical analysis
fvalue_sep, pvalue_sep = stats.f_oneway(led_05_sep,led_055_sep,led_06_sep)
fvalue_coh, pvalue_coh = stats.f_oneway(led_05_coh,led_055_coh,led_06_coh)
fvalue_align, pvalue_align = stats.f_oneway(led_05_align,led_055_align,led_06_align)
print("Research Question 1 --> Seperation F and P Value:")
print(fvalue_sep,pvalue_sep)
print("Research Question 1 --> Cohesion F and P Value:")
print(fvalue_coh,pvalue_coh)
print("Research Question 1 --> Alignment F and P Value:")
print(fvalue_align,pvalue_align)

# If pvalue < 0.05 --> Apply Tukey's Multi-Comparison Method to
# find out between which subgroups there is a significant difference

df1 = pd.DataFrame()
df1['led_05_sep'] = led_05_sep
df1['led_055_sep'] = led_055_sep
df1['led_06_sep'] = led_06_sep

df2 = pd.DataFrame()
df2['led_05_coh'] = led_05_coh
df2['led_055_coh'] = led_055_coh
df2['led_06_coh'] = led_06_coh

df3 = pd.DataFrame()
df3['led_05_align'] = led_05_align
df3['led_055_align'] = led_055_align
df3['led_06_align'] = led_06_align

# Stack the data (and rename columns):
stacked_data1 = df1.stack().reset_index()
stacked_data1 = stacked_data1.rename(columns={'level_0': 'index',
                                'level_1': 'seperation',
                                0:'violation_metric'})
stacked_data2 = df2.stack().reset_index()
stacked_data2 = stacked_data2.rename(columns={'level_0': 'index',
                                'level_1': 'cohesion',
                                0:'violation_metric'})

stacked_data3 = df3.stack().reset_index()
stacked_data3 = stacked_data3.rename(columns={'level_0': 'index',
                                'level_1': 'alignment',
                                0:'violation_metric'})

#print(stacked_data1)
```

```
MultiComp1 = MultiComparison(stacked_data1['violation_metric'],stacked_data1['
    ↪ seperation'])
MultiComp2 = MultiComparison(stacked_data2['violation_metric'],stacked_data2['cohesion
    ↪ '])
MultiComp3 = MultiComparison(stacked_data3['violation_metric'],stacked_data3['
    ↪ alignment'])

#print(MultiComp1.tukeyhsd().summary())
statistic_txt = open("statistical_test4.txt","w")
statistic_txt.write(str(MultiComp1.tukeyhsd().summary()))
statistic_txt.write("\n")
statistic_txt.write(str(MultiComp2.tukeyhsd().summary()))
statistic_txt.write("\n")
statistic_txt.write(str(MultiComp3.tukeyhsd().summary()))
statistic_txt.write("\n")
#statistic_txt.close() #to change file access modes
"""
Research Question 2 --> Relation between metrics and leader weight metric/other
    ↪ metrics
Controlled Variables: separation threshold = 0.7m , cohesion threshold = 2.25m,
alignment threshold = 45 degree, total number of robots = 13, delay = 1.8s
leader_velocity = 0.55 m/s
separation_weight = 1.0 , cohesion_weight = 1.0 and alignment_weight = 1.0
Independent variable: leader_weight = 1.0 or 1.1 or 1.2
"""
weight_10_sep = [0.174263, 0.052209, 0.395442, 0.0, 0.197315]
weight_10_coh = [0.225201, 0.360107, 0.222520, 0.538255, 0.233557]
weight_10_align = [2.076408, 2.228916, 2.268097, 2.249664, 2.146309]

weight_11_sep = [0.099063, 0.113788, 0.080429, 0.053619, 0.095174]
weight_11_coh = [0.313253, 0.195448, 0.276139, 0.237265, 0.325737]
weight_11_align = [2.275770, 2.099063, 2.298928, 2.095174, 2.064343]

weight_12_sep = [0.058981, 0.041287, 0.044177, 0.065684, 0.070415]
weight_12_coh = [0.317694, 0.197051, 0.239625, 0.241287, 0.265060]
weight_12_align = [2.353887, 2.148794, 2.167336, 2.147453, 2.271754]

bar_width = 0.25

# set height of bar
bars_sep2 = [np.mean(weight_10_sep),np.mean(weight_11_sep),np.mean(weight_12_sep)]
bars_coh2 = [np.mean(weight_10_coh),np.mean(weight_11_coh),np.mean(weight_12_coh)]
bars_align2 = [np.mean(weight_10_align),np.mean(weight_11_align),np.mean(
    ↪ weight_12_align)]

# set standard deviation of data for error bars
sep_std2 = [np.std(weight_10_sep), np.std(weight_11_sep), np.std(weight_12_sep)]
coh_std2 = [np.std(weight_10_coh), np.std(weight_11_coh), np.std(weight_12_coh)]
align_std2 = [np.std(weight_10_align), np.std(weight_11_align), np.std(weight_12_align
    ↪ )]


# Debug
print("Separation:")
print(bars_sep2)
print("Cohesion:")
```

```
print(bars_coh2)
print("Alignment:")
print(bars_align2)

# Set position of bar on X axis
r1 = np.arange(len(bars_sep2))
r2 = [x + bar_width for x in r1]
r3 = [x + bar_width for x in r2]

# Make the plot
plt.bar(r1, bars_sep2, yerr=sep_std2, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#7f6d5f', width=bar_width, edgecolor='white', label='seperation')
plt.bar(r2, bars_coh2, yerr=coh_std2, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#557f2d', width=bar_width, edgecolor='white', label='cohesion')
plt.bar(r3, bars_align2, yerr=align_std2, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#2d7f5e', width=bar_width, edgecolor='white', label='alignment')

# Add xticks on the middle of the group bars
plt.xlabel('Leader␣Weight␣over␣Other␣Boids␣Weights', fontweight='bold')
plt.ylabel('Violation␣Metrics', fontweight='bold')
plt.xticks([r + bar_width for r in range(len(bars_sep2))], ['1.0','1.1','1.2'])
plt.title('Relationship␣Between␣Relative␣Leader␣Weight␣and␣Violation␣Metrics')
# Create legend & Show graphic
plt.legend(loc=2, prop={'size': 8})
#plt.savefig('leader_weight.png')
plt.show()


# One way ANOVA analysis for statistical analysis
fvalue_sep2, pvalue_sep2 = stats.f_oneway(weight_10_sep,weight_11_sep,weight_12_sep)
fvalue_coh2, pvalue_coh2 = stats.f_oneway(weight_10_coh,weight_11_coh,weight_12_coh)
fvalue_align2, pvalue_align2 = stats.f_oneway(weight_10_align,weight_11_align,
    ↪ weight_12_align)
print("Research␣Question␣2␣-->␣Seperation␣F␣and␣P␣Value:")
print(fvalue_sep2,pvalue_sep2)
print("Research␣Question␣2␣-->␣Cohesion␣F␣and␣P␣Value:")
print(fvalue_coh2,pvalue_coh2)
print("Research␣Question␣2␣-->␣Alignment␣F␣and␣P␣Value:")
print(fvalue_align2,pvalue_align2)

# If pvalue < 0.05 --> Apply Tukey's Multi-Comparison Method to
# find out between which subgroups there is a significant difference

df4 = pd.DataFrame()
df4['weight_10_sep'] = weight_10_sep
df4['weight_11_sep'] = weight_11_sep
df4['weight_12_sep'] = weight_12_sep

df5 = pd.DataFrame()
df5['weight_10_coh'] = weight_10_coh
df5['weight_11_coh'] = weight_11_coh
df5['weight_12_coh'] = weight_12_coh

df6 = pd.DataFrame()
df6['weight_10_align'] = weight_10_align
df6['weight_11_align'] = weight_11_align
```

```
df6['weight_12_align'] = weight_12_align

# Stack the data (and rename columns):
stacked_data4 = df4.stack().reset_index()
stacked_data4 = stacked_data4.rename(columns={'level_0': 'index',
                                'level_1': 'seperation',
                                0:'violation_metric'})
stacked_data5 = df5.stack().reset_index()
stacked_data5 = stacked_data5.rename(columns={'level_0': 'index',
                                'level_1': 'cohesion',
                                0:'violation_metric'})

stacked_data6 = df6.stack().reset_index()
stacked_data6 = stacked_data6.rename(columns={'level_0': 'index',
                                'level_1': 'alignment',
                                0:'violation_metric'})

#print(stacked_data1)
MultiComp4 = MultiComparison(stacked_data4['violation_metric'],stacked_data4['
    ↪ seperation'])
MultiComp5 = MultiComparison(stacked_data5['violation_metric'],stacked_data5['cohesion
    ↪ '])
MultiComp6 = MultiComparison(stacked_data6['violation_metric'],stacked_data6['
    ↪ alignment'])

statistic_txt.write(str(MultiComp4.tukeyhsd().summary()))
statistic_txt.write("\n")
statistic_txt.write(str(MultiComp5.tukeyhsd().summary()))
statistic_txt.write("\n")
statistic_txt.write(str(MultiComp6.tukeyhsd().summary()))
statistic_txt.write("\n")
#statistic_txt.close() #to change file access modes
"""
Research Question 3 --> Relation between violation count and total number of agents
Controlled Variables: separation threshold = 0.7m , cohesion threshold = 2.25m,
alignment threshold = 45 degree, delay = 1.8s, leader_velocity = 0.55 m/s
separation_weight = 1.0 , cohesion_weight = 1.0, alignment_weight = 1.0
leader_weight = 1.0
Independent variable: total number of robots = 7 or 13 or 19
"""

total_7_sep_robot1 = [0, 0, 0, 0, 0]
total_7_sep_robot2 = [37, 83, 15, 23, 10]
total_7_sep_robot3 = [47, 69, 62, 44, 56]
total_7_sep_robot4 = [0, 20, 14, 5, 0]
total_7_sep_robot5 = [33, 92, 47, 60, 46]
total_7_sep_robot6 = [50, 226, 89, 215, 79]

total_7_coh_robot1 = [0, 0, 0, 0, 0]
total_7_coh_robot2 = [0, 0, 0, 0, 0]
total_7_coh_robot3 = [0, 0, 0, 0, 0]
total_7_coh_robot4 = [0, 0, 0, 0, 0]
total_7_coh_robot5 = [0, 0, 0, 0, 0]
total_7_coh_robot6 = [0, 0, 0, 0, 0]

total_7_align_robot1 = [120, 121, 118, 121, 123]
```

```
total_7_align_robot2 = [120, 115, 119, 120, 120]
total_7_align_robot3 = [116, 112, 116, 116, 115]
total_7_align_robot4 = [117, 121, 116, 124, 124]
total_7_align_robot5 = [118, 118, 117, 116, 116]
total_7_align_robot6 = [116, 118, 118, 117, 117]


bar_width = 0.25

# set height of bar
bars_sep3 = [np.mean(total_7_sep_robot1),np.mean(total_7_sep_robot2),np.mean(total_7_sep_robot3),
         np.mean(total_7_sep_robot4), np.mean(total_7_sep_robot5), np.mean(total_7_sep_robot6)]

bars_coh3 = [np.mean(total_7_coh_robot1),np.mean(total_7_coh_robot2),np.mean(total_7_coh_robot3),
         np.mean(total_7_coh_robot4), np.mean(total_7_coh_robot5), np.mean(total_7_coh_robot6)]

bars_align3 = [np.mean(total_7_align_robot1),np.mean(total_7_align_robot2),np.mean(total_7_align_robot3),
         np.mean(total_7_align_robot4), np.mean(total_7_align_robot5), np.mean(total_7_align_robot6)]

# set standard deviation of data for error bars
sep_std3 = [np.std(total_7_sep_robot1), np.std(total_7_sep_robot2), np.std(total_7_sep_robot3),
         np.std(total_7_sep_robot4), np.std(total_7_sep_robot5), np.std(total_7_sep_robot6)]

coh_std3 = [np.std(total_7_coh_robot1), np.std(total_7_coh_robot2), np.std(total_7_coh_robot3),
         np.std(total_7_coh_robot4), np.std(total_7_coh_robot5), np.std(total_7_coh_robot6)]

align_std3 = [np.std(total_7_align_robot1), np.std(total_7_align_robot2), np.std(total_7_align_robot3),
         np.std(total_7_align_robot4), np.std(total_7_align_robot5), np.std(total_7_align_robot6)]


# Set position of bar on X axis
r1 = np.arange(len(bars_sep3))
r2 = [x + bar_width for x in r1]
r3 = [x + bar_width for x in r2]

# Make the plot
plt.bar(r1, bars_sep3, yerr=sep_std3, error_kw=dict(lw=3, capsize=5, capthick=2), color='#7f6d5f', width=bar_width, edgecolor='white', label='seperation')
plt.bar(r2, bars_coh3, yerr=coh_std3, error_kw=dict(lw=3, capsize=5, capthick=2), color='#557f2d', width=bar_width, edgecolor='white', label='cohesion')
plt.bar(r3, bars_align3, yerr=align_std3, error_kw=dict(lw=3, capsize=5, capthick=2), color='#2d7f5e', width=bar_width, edgecolor='white', label='alignment')

# Add xticks on the middle of the group bars
```

```
plt.xlabel('Robots', fontweight='bold')
plt.ylabel('Total Violation Amount During Entire Simulation', fontweight='bold')
plt.xticks([r + bar_width for r in range(len(bars_sep3))], ['Robot1','Robot2','Robot3'
    ,'Robot4','Robot5','Robot6'])
plt.title('Total Violation Amounts for Simulation with 7 Robots')
# Create legend & Show graphic
x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,0,y2))
plt.legend(loc=9) #(loc=2, prop={'size': 8})
#plt.savefig('7robots.png')
plt.show()

####################################
total_13_sep_robot1 = [0, 0, 0, 0, 0]
total_13_sep_robot2 = [14, 0, 0, 0, 0]
total_13_sep_robot3 = [0, 0, 25, 0, 0]
total_13_sep_robot4 = [0, 0, 0, 0, 0]
total_13_sep_robot5 = [24, 16, 25, 0, 24]
total_13_sep_robot6 = [27, 12, 86, 0, 25]
total_13_sep_robot7 = [0, 0, 0, 0, 0]
total_13_sep_robot8 = [0, 0, 27, 0, 0]
total_13_sep_robot9 = [0, 0, 0, 0, 0]
total_13_sep_robot10 = [0, 0, 59, 0, 0]
total_13_sep_robot11 = [15, 0, 50, 0, 0]
total_13_sep_robot12 = [50, 11, 23, 0, 98]

total_13_coh_robot1 = [0, 0, 0, 0, 0]
total_13_coh_robot2 = [0, 0, 0, 0, 0]
total_13_coh_robot3 = [0, 0, 0, 0, 0]
total_13_coh_robot4 = [0, 0, 0, 0, 0]
total_13_coh_robot5 = [31, 61, 32, 67, 32]
total_13_coh_robot6 = [70, 71, 65, 88, 70]
total_13_coh_robot7 = [0, 0, 0, 0, 0]
total_13_coh_robot8 = [0, 0, 0, 0, 0]
total_13_coh_robot9 = [0, 0, 0, 19, 0]
total_13_coh_robot10 = [0, 0, 0, 60, 0]
total_13_coh_robot11 = [0, 64, 0, 86, 30]
total_13_coh_robot12 = [67, 73, 69, 81, 42]

total_13_align_robot1 = [107, 124, 152, 136, 129]
total_13_align_robot2 = [105, 125, 157, 107, 129]
total_13_align_robot3 = [159, 158, 168, 153, 139]
total_13_align_robot4 = [136, 155, 157, 167, 137]
total_13_align_robot5 = [127, 127, 128, 128, 133]
total_13_align_robot6 = [135, 136, 121, 134, 138]
total_13_align_robot7 = [111, 128, 128, 128, 130]
total_13_align_robot8 = [108, 129, 127, 126, 129]
total_13_align_robot9 = [164, 171, 171, 174, 131]
total_13_align_robot10 = [133, 151, 115, 137, 138]
total_13_align_robot11 = [132, 130, 137, 153, 134]
total_13_align_robot12 = [132, 131, 131, 153, 132]


bar_width = 0.25

# set height of bar
```

```
bars_sep4 = [np.mean(total_13_sep_robot1),np.mean(total_13_sep_robot2),np.mean(
    ↪ total_13_sep_robot3),
         np.mean(total_13_sep_robot4), np.mean(total_13_sep_robot5), np.mean(
            ↪ total_13_sep_robot6)]

bars_coh4 = [np.mean(total_13_coh_robot1),np.mean(total_13_coh_robot2),np.mean(
    ↪ total_13_coh_robot3),
         np.mean(total_13_coh_robot4), np.mean(total_13_coh_robot5), np.mean(
            ↪ total_13_coh_robot6)]

bars_align4 = [np.mean(total_13_align_robot1),np.mean(total_13_align_robot2),np.mean(
    ↪ total_13_align_robot3),
         np.mean(total_13_align_robot4), np.mean(total_13_align_robot5), np.mean(
            ↪ total_13_align_robot6)]

# set standard deviation of data for error bars
sep_std4 = [np.std(total_13_sep_robot1), np.std(total_13_sep_robot2), np.std(
    ↪ total_13_sep_robot3),
         np.std(total_13_sep_robot4), np.std(total_13_sep_robot5), np.std(
            ↪ total_13_sep_robot6)]

coh_std4 = [np.std(total_13_coh_robot1), np.std(total_13_coh_robot2), np.std(
    ↪ total_13_coh_robot3),
         np.std(total_13_coh_robot4), np.std(total_13_coh_robot5), np.std(
            ↪ total_13_coh_robot6)]

align_std4 = [np.std(total_13_align_robot1), np.std(total_13_align_robot2), np.std(
    ↪ total_13_align_robot3),
         np.std(total_13_align_robot4), np.std(total_13_align_robot5), np.std(
            ↪ total_13_align_robot6)]


# Set position of bar on X axis
r1 = np.arange(len(bars_sep4))
r2 = [x + bar_width for x in r1]
r3 = [x + bar_width for x in r2]

# Make the plot
plt.bar(r1, bars_sep4, yerr=sep_std4, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#7f6d5f', width=bar_width, edgecolor='white', label='seperation')
plt.bar(r2, bars_coh4, yerr=coh_std4, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#557f2d', width=bar_width, edgecolor='white', label='cohesion')
plt.bar(r3, bars_align4, yerr=align_std4, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#2d7f5e', width=bar_width, edgecolor='white', label='alignment')

# Add xticks on the middle of the group bars
plt.xlabel('Robots', fontweight='bold')
plt.ylabel('Total␣Violation␣Amount␣During␣Entire␣Simulation', fontweight='bold')
plt.xticks([r + bar_width for r in range(len(bars_sep3))], ['Robot1','Robot2','Robot3'
    ↪ ,'Robot4','Robot5','Robot6'])
plt.title('Total␣Violation␣Amounts␣for␣Simulation␣with␣13␣Robots')
# Create legend & Show graphic
x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,0,y2))
plt.legend() #(loc=2, prop={'size': 8})
#plt.savefig('13robots.png')
```

```
plt.show()

####################################
total_19_sep_robot1 = [0, 0, 0, 0, 0]
total_19_sep_robot2 = [0, 0, 0, 0, 130]
total_19_sep_robot3 = [0, 0, 0, 0, 0]
total_19_sep_robot4 = [0, 0, 0, 0, 69]
total_19_sep_robot5 = [14, 10, 19, 9, 64]
total_19_sep_robot6 = [0, 0, 0, 35, 18]
total_19_sep_robot7 = [0, 0, 0, 0, 119]
total_19_sep_robot8 = [0, 177, 0, 8, 100]
total_19_sep_robot9 = [0, 0, 0, 0, 0]
total_19_sep_robot10 = [0, 200, 0, 0, 57]
total_19_sep_robot11 = [0, 0, 0, 0, 0]
total_19_sep_robot12 = [9, 0, 0, 12, 5]
total_19_sep_robot13 = [0, 0, 0, 30, 4]
total_19_sep_robot14 = [0, 0, 0, 0, 0]
total_19_sep_robot15 = [0, 0, 0, 43, 0]
total_19_sep_robot16 = [0, 0, 0, 27, 0]
total_19_sep_robot17 = [18, 46, 31, 40, 22]
total_19_sep_robot18 = [0, 140, 0, 64, 64]

total_19_coh_robot1 = [187, 0, 226, 0, 0]
total_19_coh_robot2 = [0, 0, 47, 0, 0]
total_19_coh_robot3 = [17, 8, 19, 6, 9]
total_19_coh_robot4 = [28, 30, 53, 51, 0]
total_19_coh_robot5 = [64, 64, 72, 64, 0]
total_19_coh_robot6 = [269, 74, 229, 75, 74]
total_19_coh_robot7 = [177, 0, 186, 0, 0]
total_19_coh_robot8 = [0, 0, 0, 0, 0]
total_19_coh_robot9 = [175, 21, 191, 0, 0]
total_19_coh_robot10 = [53, 0, 16, 30, 0]
total_19_coh_robot11 = [278, 70, 232, 68, 68]
total_19_coh_robot12 = [82, 85, 87, 82, 83]
total_19_coh_robot13 = [216, 0, 128, 0, 0]
total_19_coh_robot14 = [187, 0, 238, 0, 0]
total_19_coh_robot15 = [159, 73, 165, 66, 71]
total_19_coh_robot16 = [221, 75, 243, 69, 73]
total_19_coh_robot17 = [81, 80, 85, 79, 81]
total_19_coh_robot18 = [154, 142, 151, 95, 112]

total_19_align_robot1 = [137, 140, 151, 128, 143]
total_19_align_robot2 = [137, 148, 152, 141, 137]
total_19_align_robot3 = [143, 175, 148, 112, 182]
total_19_align_robot4 = [175, 179, 174, 177, 165]
total_19_align_robot5 = [143, 149, 136, 165, 168]
total_19_align_robot6 = [146, 193, 145, 143, 145]
total_19_align_robot7 = [138, 144, 140, 115, 164]
total_19_align_robot8 = [135, 156, 166, 137, 131]
total_19_align_robot9 = [172, 178, 167, 180, 173]
total_19_align_robot10 = [166, 141, 160, 182, 175]
total_19_align_robot11 = [141, 139, 186, 118, 141]
total_19_align_robot12 = [141, 142, 142, 119, 141]
total_19_align_robot13 = [147, 136, 134, 143, 166]
total_19_align_robot14 = [143, 133, 176, 159, 140]
total_19_align_robot15 = [189, 164, 190, 153, 166]
```

```
total_19_align_robot16 = [169, 154, 158, 145, 150]
total_19_align_robot17 = [189, 166, 202, 142, 145]
total_19_align_robot18 = [147, 140, 189, 141, 142]


bar_width = 0.25

# set height of bar
bars_sep5 = [np.mean(total_19_sep_robot1),np.mean(total_19_sep_robot2),np.mean(
    ↪ total_19_sep_robot3),
         np.mean(total_19_sep_robot4), np.mean(total_19_sep_robot5), np.mean(
             ↪ total_19_sep_robot6)]

bars_coh5 = [np.mean(total_19_coh_robot1),np.mean(total_19_coh_robot2),np.mean(
    ↪ total_19_coh_robot3),
         np.mean(total_19_coh_robot4), np.mean(total_19_coh_robot5), np.mean(
             ↪ total_19_coh_robot6)]

bars_align5 = [np.mean(total_19_align_robot1),np.mean(total_19_align_robot2),np.mean(
    ↪ total_19_align_robot3),
         np.mean(total_19_align_robot4), np.mean(total_19_align_robot5), np.mean(
             ↪ total_19_align_robot6)]

# set standard deviation of data for error bars
sep_std5 = [np.std(total_19_sep_robot1), np.std(total_19_sep_robot2), np.std(
    ↪ total_19_sep_robot3),
         np.std(total_19_sep_robot4), np.std(total_19_sep_robot5), np.std(
             ↪ total_19_sep_robot6)]

coh_std5 = [np.std(total_19_coh_robot1), np.std(total_19_coh_robot2), np.std(
    ↪ total_19_coh_robot3),
         np.std(total_19_coh_robot4), np.std(total_19_coh_robot5), np.std(
             ↪ total_19_coh_robot6)]

align_std5 = [np.std(total_19_align_robot1), np.std(total_19_align_robot2), np.std(
    ↪ total_19_align_robot3),
         np.std(total_19_align_robot4), np.std(total_19_align_robot5), np.std(
             ↪ total_19_align_robot6)]


# Set position of bar on X axis
r1 = np.arange(len(bars_sep5))
r2 = [x + bar_width for x in r1]
r3 = [x + bar_width for x in r2]

# Make the plot
plt.bar(r1, bars_sep5, yerr=sep_std5, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#7f6d5f', width=bar_width, edgecolor='white', label='seperation')
plt.bar(r2, bars_coh5, yerr=coh_std5, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#557f2d', width=bar_width, edgecolor='white', label='cohesion')
plt.bar(r3, bars_align5, yerr=align_std5, error_kw=dict(lw=3, capsize=5, capthick=2),
    ↪ color='#2d7f5e', width=bar_width, edgecolor='white', label='alignment')

# Add xticks on the middle of the group bars
plt.xlabel('Robots', fontweight='bold')
plt.ylabel('Total Violation Amount During Entire Simulation', fontweight='bold')
```

```python
plt.xticks([r + bar_width for r in range(len(bars_sep5))], ['Robot1','Robot2','Robot3','Robot4','Robot5','Robot6'])
plt.title('Total Violation Amounts for Simulation with 19 Robots')
# Create legend & Show graphic
x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,0,y2))
plt.legend() #(loc=2, prop={'size': 8})
#plt.savefig('19robots.png')
plt.show()


# One way ANOVA analysis for statistical analysis
total_7_sep = (total_7_sep_robot1 + total_7_sep_robot2 + total_7_sep_robot3 +
           total_7_sep_robot4 + total_7_sep_robot5 + total_7_sep_robot6)
total_7_coh = (total_7_coh_robot1 + total_7_coh_robot2 + total_7_coh_robot3 +
           total_7_coh_robot4 + total_7_coh_robot5 + total_7_coh_robot6)
total_7_align = (total_7_align_robot1 + total_7_align_robot2 + total_7_align_robot3 +
           total_7_align_robot4 + total_7_align_robot5 + total_7_align_robot6)

total_13_sep = (total_13_sep_robot1 + total_13_sep_robot2 + total_13_sep_robot3 +
           total_13_sep_robot4 + total_13_sep_robot5 + total_13_sep_robot6)
total_13_coh = (total_13_coh_robot1 + total_13_coh_robot2 + total_13_coh_robot3 +
           total_13_coh_robot4 + total_13_coh_robot5 + total_13_coh_robot6)
total_13_align = (total_13_align_robot1 + total_13_align_robot2 + total_13_align_robot3 +
           total_13_align_robot4 + total_13_align_robot5 + total_13_align_robot6)


total_19_sep = (total_19_sep_robot1 + total_19_sep_robot2 + total_19_sep_robot3 +
           total_19_sep_robot4 + total_19_sep_robot5 + total_19_sep_robot6)
total_19_coh = (total_19_coh_robot1 + total_19_coh_robot2 + total_19_coh_robot3 +
           total_19_coh_robot4 + total_19_coh_robot5 + total_19_coh_robot6)
total_19_align = (total_19_align_robot1 + total_19_align_robot2 + total_19_align_robot3 +
           total_19_align_robot4 + total_19_align_robot5 + total_19_align_robot6)



fvalue_sep3, pvalue_sep3 = stats.f_oneway(total_7_sep,total_13_sep,total_19_sep)
fvalue_coh3, pvalue_coh3 = stats.f_oneway(total_7_coh,total_13_coh,total_19_coh)
fvalue_align3, pvalue_align3 = stats.f_oneway(total_7_align,total_13_align,total_19_align)
print("Research Question 3 --> Seperation F and P Value:")
print(fvalue_sep3,pvalue_sep3)
print("Research Question 3 --> Cohesion F and P Value:")
print(fvalue_coh3,pvalue_coh3)
print("Research Question 3 --> Alignment F and P Value:")
print(fvalue_align3,pvalue_align3)

# If pvalue < 0.05 --> Apply Tukey's Multi-Comparison Method to
# find out between which subgroups there is a significant difference

df7 = pd.DataFrame()
df7['total_7_sep'] = total_7_sep
df7['total_13_sep'] = total_13_sep
df7['total_19_sep'] = total_19_sep
```

```
df8 = pd.DataFrame()
df8['total_7_coh'] = total_7_coh
df8['total_13_coh'] = total_13_coh
df8['total_19_coh'] = total_19_coh

df9 = pd.DataFrame()
df9['total_7_align'] = total_7_align
df9['total_13_align'] = total_13_align
df9['total_19_align'] = total_19_align

# Stack the data (and rename columns):
stacked_data7 = df7.stack().reset_index()
stacked_data7 = stacked_data7.rename(columns={'level_0': 'index',
                                  'level_1': 'seperation',
                                  0:'total_violation'})
stacked_data8 = df8.stack().reset_index()
stacked_data8 = stacked_data8.rename(columns={'level_0': 'index',
                                  'level_1': 'cohesion',
                                  0:'total_violation'})

stacked_data9 = df9.stack().reset_index()
stacked_data9 = stacked_data9.rename(columns={'level_0': 'index',
                                  'level_1': 'alignment',
                                  0:'total_violation'})

#print(stacked_data1)
MultiComp7 = MultiComparison(stacked_data7['total_violation'],stacked_data7['seperation'])
MultiComp8 = MultiComparison(stacked_data8['total_violation'],stacked_data8['cohesion'])
MultiComp9 = MultiComparison(stacked_data9['total_violation'],stacked_data9['alignment'])

statistic_txt.write(str(MultiComp7.tukeyhsd().summary()))
statistic_txt.write("\n")
statistic_txt.write(str(MultiComp8.tukeyhsd().summary()))
statistic_txt.write("\n")
statistic_txt.write(str(MultiComp9.tukeyhsd().summary()))
statistic_txt.write("\n")
statistic_txt.close() #to change file access modes
```

D. leader_controller.py

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist, Pose
from nav_msgs.msg import Odometry

from math import pow, atan2, sqrt

class Leader:

   def __init__(self):
      self.velocity_publisher = rospy.Publisher('/robot_0/cmd_vel', Twist, queue_size=10)
```

```python
        rospy.init_node('leader_controller', anonymous=True)
        self.odom_subscriber = rospy.Subscriber('/robot_0/odom', Odometry, self.
            ↪ update_leader_pose)

        #self.speed = rospy.get_param("/dyn_reconf/max_speed")
        self.pose = Pose()
        self.rate = rospy.Rate(10)

    def update_leader_pose(self,data):
        self.pose = data.pose.pose

    def calculate_distance(self, goal_pose):
        return sqrt(pow((goal_pose.position.x - self.pose.position.x), 2)
                 + pow((goal_pose.position.y - self.pose.position.y), 2))

    def linear_vel(self, goal_pose):
        Kp_lin = 1.5
        return Kp_lin * self.calculate_distance(goal_pose)

    def steering_angle(self, goal_pose):
        return atan2(goal_pose.position.y - self.pose.position.y,
                  goal_pose.position.x - self.pose.position.x)

    def angular_vel(self, goal_pose):
        Kp_ang = 6
        current_angle = atan2(self.pose.position.y,self.pose.position.x)
        return Kp_ang * (self.steering_angle(goal_pose) - current_angle)

    def go_desired_pose(self,desired_pose):
        target_pose = Pose()
        target_pose.position.x = 0.0 #desired_pose.position.x
        target_pose.position.y = 3.0 #desired_pose.position.y

        error_margin = 0.1
        vel_msg = Twist()
        while self.calculate_distance(target_pose) >= error_margin:
            vel_msg.linear.x = self.linear_vel(target_pose)
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0

            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            vel_msg.angular.z = self.angular_vel(target_pose)

            self.velocity_publisher.publish(vel_msg)

            self.rate.sleep() #publish at the desired rate


        #Stop the leader after movement is done
        vel_msg.linear.x = 0
        vel_msg.angular.z = 0
        self.velocity_publisher.publish(vel_msg)

        rospy.spin()
```

```
def move_forward(self,amount,speed):
    target_pose = Pose()
    target_pose.position.x = self.pose.position.x #desired_pose.position.x
    target_pose.position.y = amount #desired_pose.position.y

    error_margin = 0.1
    vel_msg = Twist()
    while self.calculate_distance(target_pose) >= error_margin:
        vel_msg.linear.x = 0
        vel_msg.linear.y = speed
        vel_msg.linear.z = 0

        vel_msg.angular.x = 0
        vel_msg.angular.y = 0
        vel_msg.angular.z = 0

        self.velocity_publisher.publish(vel_msg)

        self.rate.sleep() #publish at the desired rate


    #Stop the leader after movement is done
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    self.velocity_publisher.publish(vel_msg)

    #rospy.spin()


def move_backward(self,amount,speed):
    target_pose = Pose()
    target_pose.position.x = self.pose.position.x #desired_pose.position.x
    target_pose.position.y = -amount #desired_pose.position.y

    error_margin = 0.1
    vel_msg = Twist()
    while self.calculate_distance(target_pose) >= error_margin:
        vel_msg.linear.x = 0
        vel_msg.linear.y = -speed
        vel_msg.linear.z = 0

        vel_msg.angular.x = 0
        vel_msg.angular.y = 0
        vel_msg.angular.z = 0

        self.velocity_publisher.publish(vel_msg)

        self.rate.sleep() #publish at the desired rate


    #Stop the leader after movement is done
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    self.velocity_publisher.publish(vel_msg)
```

```python
    #rospy.spin()


def move_right(self,amount,speed):
    target_pose = Pose()
    target_pose.position.x = amount #desired_pose.position.x
    target_pose.position.y = self.pose.position.y #desired_pose.position.y

    error_margin = 0.1
    vel_msg = Twist()
    while self.calculate_distance(target_pose) >= error_margin:
        vel_msg.linear.x = speed
        vel_msg.linear.y = 0
        vel_msg.linear.z = 0

        vel_msg.angular.x = 0
        vel_msg.angular.y = 0
        vel_msg.angular.z = 0

        self.velocity_publisher.publish(vel_msg)

        self.rate.sleep() #publish at the desired rate


    #Stop the leader after movement is done
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    self.velocity_publisher.publish(vel_msg)

    #rospy.spin()

def move_left(self,amount,speed):
    target_pose = Pose()
    target_pose.position.x = -amount #desired_pose.position.x
    target_pose.position.y = self.pose.position.y #desired_pose.position.y
    error_margin = 0.1
    vel_msg = Twist()
    while self.calculate_distance(target_pose) >= error_margin:
        vel_msg.linear.x = -speed
        vel_msg.linear.y = 0
        vel_msg.linear.z = 0

        vel_msg.angular.x = 0
        vel_msg.angular.y = 0
        vel_msg.angular.z = 0

        self.velocity_publisher.publish(vel_msg)

        self.rate.sleep() #publish at the desired rate


    #Stop the leader after movement is done
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    self.velocity_publisher.publish(vel_msg)
```

```
        #rospy.spin()


    def draw_square(self):
        speed = 0.55
        self.move_forward(4.0,speed)
        self.move_left(4.0,speed)
        self.move_backward(4.0,speed)
        self.move_right(4.0,speed)
        self.move_forward(4.0,speed)
        self.move_left(4.0,speed)

        rospy.spin()



if __name__ == '__main__':
    try:
        leader = Leader()
        leader.draw_square()
    except rospy.ROSInterruptException: pass
```

CODE WRITTEN INITIALLY BY Marko Križmančić, BUT THEN LATER MODIFIED BY ME:

E. dynamic_reconfiguration_node.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import rospy
from dynamic_reconfigure.server import Server
from boids_ros.cfg import ReynoldsConfig


class DynReconf():
    """
    Dynamic reconfigure server.

    Process parameter changes from rqt_reconfigure and update parameter server.
    Publish empty message to let other nodes know there are updated parameters
    on server.
    """
    def __init__(self):
        """Initialize dynamic reconfigure server."""
        Server(ReynoldsConfig, self.callback)

        # Keep program from exiting
        rospy.spin()

    def callback(self, config, level):
        """Display all parameters when changed and signal to update."""
        rospy.loginfo("[Dynamic␣reconfigure]␣=>␣\n" +
                      """\tReconfigure Request:
                        Alignment: {alignment_weight}
                        Cohesion: {cohesion_weight}
                        Separation: {separation_weight}
```

```
                    Obstacle: {obstacle_weight}
                    Leader: {leader_weight}
                    Max speed: {max_speed}
                    Max force: {max_force}
                    Friction: {friction}
                    Desired seperation: {desired_separation}
                    Horizon: {horizon}
                    Obstacle radius: {avoid_radius}""".format(**config))
        return config


if __name__ == "__main__":
    # Initialize the node and name it.
    rospy.init_node("dyn_reconf", anonymous=False)

    # Go to class functions that do all the heavy lifting.
    # Do error checking.
    try:
        dr = DynReconf()
    except rospy.ROSInterruptException:
        pass
```

F. boids.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import math
import rospy
from geometry_msgs.msg import Twist, PoseStamped, Pose
from util import Vector2, angle_diff


class Boid(object):
    def __init__(self, initial_velocity_x, initial_velocity_y, wait_count, start_count,
        ↪ frequency):
        self.position = Vector2()
        self.velocity = Vector2()
        self.mass = 0.18 # Mass of Sphero robot in kilograms
        self.wait_count = wait_count # Waiting time before starting
        self.start_count = start_count # Time during which initial velocity is being
            ↪ sent
        self.frequency = frequency # Control loop frequency

        # Set initial velocity
        self.initial_velocity = Twist()
        self.initial_velocity.linear.x = initial_velocity_x
        self.initial_velocity.linear.y = initial_velocity_y

        # This dictionary holds values of each flocking components and is used
        # to pass them to the visualization markers publisher.
        self.viz_components = {}

    def update_parameters(self, params):
        self.rule1_weight = params['cohesion_weight']
        self.rule2_weight = params['separation_weight']
```

```
        self.rule3_weight = params['alignment_weight']
        self.obstacle_weight = params['obstacle_weight']
        #self.leader_weight = 1.0
        self.leader_weight = params['leader_weight']
        self.max_speed = params['max_speed']
        self.max_force = params['max_force']
        self.friction = params['friction']
        self.desired_separation = params['desired_separation']
        self.horizon = params['horizon']
        self.avoid_radius = params['avoid_radius']


    def rule1(self, nearest_agents): #Cohesion
        center_of_mass = Vector2()
        com_direction = Vector2()
        # Find mean position of neighboring agents.
        for b in nearest_agents:
            boid_position = get_agent_position(b)
            center_of_mass += boid_position

        # Magnitude of force is proportional to agents' distance
        # from the center of mass.
        # Force should be applied in the direction of com
        if nearest_agents:
            com_direction = center_of_mass / len(nearest_agents)
            #rospy.logdebug("cohesion*: %s", direction)
            d = com_direction.norm()
            com_direction.set_mag((self.max_force * (d / self.horizon)))

        return com_direction

    def rule2(self, nearest_agents): #Seperation

        c = Vector2()
        N = 0 #Total boid number

        for b in nearest_agents:
            boid_position = get_agent_position(b)
            d = boid_position.norm()
            if d < self.desired_separation:
                N += 1
                boid_position *= -1 # Force towards outside
                boid_position.normalize() # Normalize to get only direction.
                # magnitude is proportional to inverse square of d
                # where d is the distance between agents
                boid_position = boid_position / (d**2)
                c += boid_position

        if N:
            c /= N #average
            c.limit(2 * self.max_force) # 2 * max_force gives this rule a slight priority
                ↪ .

        return c

    def rule3(self, nearest_agents): #Alignment
```

```
    perceived_velocity = Vector2()
    pv = Vector2()
    # Find mean direction of neighboring agents.
    for boid in nearest_agents:
        boid_velocity = get_agent_velocity(boid)
        perceived_velocity += boid_velocity #mean perceived

    # Steer toward calculated mean direction with maximum velocity.
    if nearest_agents:
        perceived_velocity.set_mag(self.max_speed)
        pv = perceived_velocity - self.velocity
        pv.limit(self.max_force)
    return pv


def compute_leader_following(self,rel2leader):
    for agent in rel2leader:
        rel2leader_position = get_leader_position(agent)
        # Force in the direction that minimizes rel_position 2 leader
        # i.e. it should be in the direction of rel2leader
        direction = Vector2() #initiliazes (0,0)
        direction = rel2leader_position # *0.01
        # d = direction.norm()
        # direction.set_mag((self.max_force * d))

    return direction


def compute_velocity(self, my_agent, nearest_agents,rel2leader):
    """Compute total velocity based on all components."""

    # While waiting to start, send zero velocity and decrease counter.
    if self.wait_count > 0:
        self.wait_count -= 1
        rospy.logdebug("wait␣" + '{}'.format(self.wait_count))
        rospy.logdebug("velocity:\n%s", Twist().linear)
        return Twist(), None

    # Send initial velocity and decrease counter.
    elif self.start_count > 0:
        self.start_count -= 1
        rospy.logdebug("start␣" + '{}'.format(self.start_count))
        rospy.logdebug("velocity:\n%s", self.initial_velocity.linear)
        return self.initial_velocity, None

    # Normal operation, velocity is determined using Reynolds' rules.
    else:
        self.velocity = get_agent_velocity(my_agent)
        self.old_heading = self.velocity.arg()
        self.old_velocity = Vector2(self.velocity.x, self.velocity.y)
        rospy.logdebug("old_velocity:␣%s", self.velocity)

        # Compute all the components.
        v1 = self.rule1(nearest_agents) #cohesion
        v2 = self.rule2(nearest_agents) #seperation
        v3 = self.rule3(nearest_agents) #alignment
```

```
        leader = self.compute_leader_following(rel2leader)


        # Add components together and limit the output.
        force = Vector2()
        force += v1 * self.rule1_weight
        force += v2 * self.rule2_weight
        force += v3 * self.rule3_weight
        force += leader * self.leader_weight

        force.limit(self.max_force)

        # If agent is moving, apply constant friction force.
        # If agent's velocity is less then friction / 2, it would get
        # negative velocity. In this case, just stop it.
        #if self.velocity.norm() > self.friction / 2:
        # force += self.friction * -1 * self.velocity.normalize(ret=True)
        #else:
        # self.velocity = Vector2()

        acceleration = force / self.mass

        # Calculate total velocity (delta_velocity = acceleration * delta_time).
        self.velocity += acceleration / self.frequency
        self.velocity.limit(self.max_speed)

        #rospy.logdebug("force: %s", force)
        #rospy.logdebug("acceleration: %s", acceleration / self.frequency)
        #rospy.logdebug("velocity: %s\n", self.velocity)

        # Return the the velocity as Twist message.
        vel = Twist()
        vel.linear.x = self.velocity.x
        vel.linear.y = self.velocity.y

        # Pack all components for Rviz visualization.
        # Make sure these keys are the same as the ones in `util.py`.
        self.viz_components['cohesion'] = v1 * self.rule1_weight
        self.viz_components['separation'] = v2 * self.rule2_weight
        self.viz_components['alignment'] = v3 * self.rule3_weight
        #self.viz_components['avoid'] = avoid * self.obstacle_weight
        self.viz_components['leader'] = leader * self.leader_weight
        #self.viz_components['acceleration'] = acceleration / self.frequency
        #self.viz_components['velocity'] = self.velocity
        #self.viz_components['estimated'] = self.old_velocity
        return vel, self.viz_components

def get_agent_velocity(agent):
   vel = Vector2()
   vel.x = agent.twist.twist.linear.x
   vel.y = agent.twist.twist.linear.y
   return vel


def get_agent_position(agent):
```

```python
    pos = Vector2()
    pos.x = agent.pose.pose.position.x
    pos.y = agent.pose.pose.position.y
    return pos

def get_leader_position(leader):
    pos = Vector2()
    pos.x = leader.position.x
    pos.y = leader.position.y
    return pos


def get_obst_position(obst):
    pos = Vector2()
    pos.x = obst.position.x
    pos.y = obst.position.y
    return pos
```

G. nearest_search.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import print_function
import math
import rospy
import message_filters as mf
from copy import deepcopy

from dynamic_reconfigure.msg import Config
from geometry_msgs.msg import PoseArray, Pose, PoseStamped
from nav_msgs.msg import Odometry, OccupancyGrid
from boids_ros.msg import OdometryArray

class NearestSearch(object):
    """
    Node that provides information about nearest flockmates and obstacles.

    Generally, Reynolds' flocking algorithm works on distributed systems.
    If agents don't have any sensors, a centralized system is needed. This node
    is an 'all-knowing' hub that makes virtual distributed system possible.
    It is subscribed to messages with position and velocity of each agent and
    knows the map layout. For each agent, it finds its neighbors within search
    radius and calculates their relative position. This data is then published
    to individual agents along side the list of obstacles within range.
    """

    def map_callback(self, data):
        """Save map occupancy grid and meta-data in class variables."""
        self.map = []
        self.map_width = data.info.width
        self.map_height = data.info.height
        self.map_resolution = data.info.resolution
        self.map_origin = data.info.origin.position

        # Reverse the order of rows in map
```

```
        for i in range(self.map_height - 1, -1, -1):
            self.map.append(data.data[i * self.map_width:(i + 1) * self.map_width])

    def param_callback(self, data):
        """Update search parameters from server."""
        while not rospy.has_param('/dyn_reconf/horizon'):
            rospy.sleep(0.1)

        self.horizon = rospy.get_param('/dyn_reconf/horizon')
        self.r = int(self.horizon / self.map_resolution)

    def robot_callback(self, *data):
        """
        This callback function is used to publish following topics:
            nearest_robots --> contains pose of nearby agents to each agent in flock
        e.g. /robot_1/robots topic contains pose of nearby agents to robot_1
            avoids --> contains pose of obstacles in the map relative to each agent
        e.g. /robot_2/avoids topic contains pose of obstacles rel. to robot_2
            rel_target --> contains pose of each agent relative to the leader (robot_0)
        e.g. /robot_6/rel2leader topic contains relative pose of robot_6 to leader
        Note: For simplicity, robot_0 is chosen as a leader!
        """

        for robot in data:
            time = rospy.Time.now() #Current time
            robot_name = robot.header.frame_id.split('/')[1] #robot name

            robot_position = robot.pose.pose.position #current robot's position

            ####################### Nearest Robots ###################################
            nearest_robots = OdometryArray() #collect pose of all nearby robots
            nearest_robots.header.stamp = time

            nearest_robots.array.append(deepcopy(robot)) # add current robot's odom to
                ↪ array

            # Now look for neighbor robots within horizon of each robot
            for neighbor in data:
                neighbor_position = neighbor.pose.pose.position
                # Distance between robot_position and neighbor_position
                d = math.sqrt(pow(robot_position.x - neighbor_position.x, 2)
                          + pow(robot_position.y - neighbor_position.y, 2))
                if d > 0 and d <= self.horizon:
                    rel_neighbor_pos = deepcopy(neighbor)
                    rel_neighbor_pos.pose.pose.position.x = neighbor_position.x -
                        ↪ robot_position.x
                    rel_neighbor_pos.pose.pose.position.y = neighbor_position.y -
                        ↪ robot_position.y
                    nearest_robots.array.append(rel_neighbor_pos)

            self.nearest[robot_name].publish(nearest_robots) #Send to ros publisher

            ##########################################################################

            ############################## Leader ################################
            rel_target = PoseArray() #contains pose of current agent rel to leader
```

```
        if robot_name == "robot_0": #leader
            leader = Pose() #contains pose of leader
            leader.position = robot.pose.pose.position
        else:
            pose_target = Pose()
            pose_target.position.x = leader.position.x-robot_position.x
            pose_target.position.y = leader.position.y-robot_position.y
            # getting time to rel_target header is important to synchorize this topic
            # with other ROS topics (e.g. .../obstacles .../nearest_robots)
            rel_target.header.stamp = time #this is important to synchorize
            rel_target.poses.append(pose_target)
            self.leader[robot_name].publish(rel_target)
        ##################################################################


    def __init__(self):
        """Create subscribers and publishers."""

        # Get parameters and initialize class variables.
        self.num_agents = rospy.get_param('/num_of_robots')
        robot_name = rospy.get_param('~robot_name')

        # Create publishers for commands
        pub_keys = [robot_name + '_{}'.format(i) for i in range(self.num_agents)]

        # Publisher for locations of nearest agents
        self.nearest = dict.fromkeys(pub_keys)
        for key in self.nearest.keys():
            self.nearest[key] = rospy.Publisher('/' + key + '/nearest', OdometryArray,
                ↪ queue_size=1)

        # Publisher for relative position to leader
        self.leader = dict.fromkeys(pub_keys)
        for key in self.leader.keys():
            # robot_0 is leader, so no need to publish it's relative position to itself
                ↪ :)
            if key != "robot_0":
                self.leader[key] = rospy.Publisher('/' + key + '/rel2leader', PoseArray,
                    ↪ queue_size=1)

        # Create subscribers
        rospy.Subscriber('/map', OccupancyGrid, self.map_callback, queue_size=1)
        rospy.sleep(0.5) # Wait for first map_callback to finish
        rospy.Subscriber('/dyn_reconf/parameter_updates', Config, self.param_callback,
            ↪ queue_size=1)
        self.param_callback(None)

        topic_name = '/' + robot_name + '_{}/odom'

        subs = [mf.Subscriber(topic_name.format(i), Odometry) for i in range(self.
            ↪ num_agents)]
        self.ts = mf.ApproximateTimeSynchronizer(subs, 10, 0.11)
        self.ts.registerCallback(self.robot_callback)

        rospy.spin()
```

```python
if __name__ == '__main__':
    rospy.init_node('NearestSearch')

    try:
        ns = NearestSearch()
    except rospy.ROSInterruptException:
        pass
```

H. reynolds_controller.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import rospy
import message_filters as mf
from dynamic_reconfigure.msg import Config
from geometry_msgs.msg import Twist, PoseArray, PoseStamped, Pose
from visualization_msgs.msg import MarkerArray

from boids import Boid
from util import MarkerSet
from boids_ros.msg import OdometryArray


class ReynoldsController(object):
    """
    ROS node implementation of Reynolds' flocking algorithm.

    This node represents a single agent in the flock. It subscribes to the list
    of other agents within search radius. Velocity of the agent is calculated
    based on Reynolds' flocking rules and this information is published to the
    simulator or physical implementation of the agent.
    """

    def callback(self, *data):
        """
        Unpack received data, compute velocity and publish the result.

        This is a callback of message_filters TimeSynchronizer subscriber. It is
        called only when all defined messages arrive with the same time stamp.
        In this case, there are two messages: "nearest" of type OdometryArray
        and "avoid" of type PoseArray. `data` is a list containing data from
        these messages.
            `data[0]` contains neighboring agents
            `data[1]` contains positions of obstacles
        """
        my_agent = data[0].array[0] # odometry data for this agent is first in list
        nearest_agents = data[0].array[1:] # odometry data for neighbors follows
        rel2leader = data[1].poses # relative position to leader

        if self.params_set:
            # Compute agent's velocity and publish the command.
            ret_vel, viz = self.agent.compute_velocity(my_agent, nearest_agents,
                ↪ rel2leader)
            average_heading = self.markers.get_heading(viz)
```

```
        # This is for use with real robots (Spheros).
        if self.run_type == 'real':
            cmd_vel = Twist()
            cmd_vel.linear.x = int(ret_vel.linear.x * 100)
            cmd_vel.linear.y = int(ret_vel.linear.y * 100)
            self.cmd_vel_pub.publish(cmd_vel)
        # This is for use with simulation.
        elif self.run_type == 'sim':
            self.cmd_vel_pub.publish(ret_vel)

        # Publish markers for visualization in Rviz.
        marker_array = self.markers.update_data(viz)

        self.markers_pub.publish(marker_array)

    def param_callback(self, data):
        """Call method for updating flocking parameters from server."""
        param_names = ['alignment_weight', 'cohesion_weight', 'separation_weight', '
            ↪ obstacle_weight',
                       'leader_weight', 'max_speed', 'max_force', 'friction', '
                           ↪ desired_separation',
                       'horizon', 'avoid_radius']
        # Dictionary for passing parameters.
        param_dict = {param: rospy.get_param('/dyn_reconf/' + param) for param in
            ↪ param_names}
        self.agent.update_parameters(param_dict)
        self.params_set = True

    def __init__(self):
        """Initialize agent instance, create subscribers and publishers."""
        # Initialize class variables.
        init_vel_x = rospy.get_param("~init_vel_x", 0)
        init_vel_y = rospy.get_param("~init_vel_y", 0)
        frequency = rospy.get_param("/ctrl_loop_freq")
        wait_count = int(rospy.get_param("/wait_time") * frequency)
        start_count = int(rospy.get_param("/start_time") * frequency)
        self.run_type = rospy.get_param("/run_type")
        self.enable_leader_following = rospy.get_param("/enable_leader_following")
        self.agent = Boid(init_vel_x, init_vel_y, wait_count, start_count, frequency)
        self.markers = MarkerSet()
        self.params_set = False

        # Create a publisher for commands.
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=frequency)
        self.markers_pub = rospy.Publisher('markers', MarkerArray, queue_size=frequency)

        # Create subscribers.
        rospy.Subscriber('/dyn_reconf/parameter_updates', Config, self.param_callback,
            ↪ queue_size=1)
        #rospy.Subscriber("leader", PoseStamped, self.leader_callback,queue_size=1)

        subs = [mf.Subscriber("nearest", OdometryArray), mf.Subscriber("rel2leader",
            ↪ PoseArray)]
        self.ts = mf.TimeSynchronizer(subs, 10)
        self.ts.registerCallback(self.callback)
```

```
        # Keep program from exiting
        rospy.spin()


if __name__ == '__main__':
    # Initialize the node and name it.
    rospy.init_node('ReynoldsController')#, log_level=rospy.DEBUG)

    # Go to class functions that do all the heavy lifting
    # Do error checking
    try:
        rc = ReynoldsController()
    except rospy.ROSInterruptException:
        pass
```

I. util.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
This module is used for utility and helper functions.

Classes:
    Vector2: 2D vector class representation with x and y components
    MarkerSet: convenience class for handling interactive Rviz markers

Function:
    pose_dist: calculate distance between two ROS Pose type variables
"""

import math
import rospy
import logging
import numpy as np
from visualization_msgs.msg import Marker, MarkerArray
from std_msgs.msg import ColorRGBA
from geometry_msgs.msg import Pose, Vector3, Quaternion
from tf.transformations import quaternion_from_euler


class Vector2(object):
    """
    2D vector class representation with x and y components.

    Supports simple addition, subtraction, multiplication, division and
    normalization, as well as getting norm and angle of the vector and
    setting limit and magnitude.

    Attributes:
        x (float): x component of the vector
        y (float): y component of the vector

    Methods:
        norm(self): Return the norm of the vector
```

```
        arg(self): Return the angle of the vector
        normalize(self): Normalize the vector
        limit(self, value): Limit vector’s maximum magnitude to given value
        set_mag(self, value): Set vector’s magnitude without changing direction
    """

    def __init__(self, x=0, y=0):
        """
        Initialize vector components.

        Args:
            x (float): x component of the vector
            y (float): y component of the vector
        """
        self.x = x
        self.y = y

    @classmethod
    def from_norm_arg(cls, norm=0, arg=0):
        inst = cls(1, 1)
        inst.set_mag(norm)
        inst.set_angle(arg)
        return inst

    def __add__(self, other):
        if isinstance(other, self.__class__):
            return Vector2(self.x + other.x, self.y + other.y)
        elif isinstance(other, int) or isinstance(other, float):
            return Vector2(self.x + other, self.y + other)

    def __sub__(self, other):
        if isinstance(other, self.__class__):
            return Vector2(self.x - other.x, self.y - other.y)
        elif isinstance(other, int) or isinstance(other, float):
            return Vector2(self.x - other, self.y - other)

    def __div__(self, other):
        if isinstance(other, self.__class__):
            raise ValueError("Cannot divide two vectors!")
        elif isinstance(other, int) or isinstance(other, float):
            if other != 0:
                return Vector2(self.x / other, self.y / other)
            else:
                return Vector2()

    def __mul__(self, other):
        if isinstance(other, self.__class__):
            raise NotImplementedError("Multiplying vectors is not implemented!")
        elif isinstance(other, int) or isinstance(other, float):
            return Vector2(self.x * other, self.y * other)

    def __rmul__(self, other):
        return self.__mul__(other)

    def __str__(self):
        return "({: .5f}, {: 6.1f})".format(self.norm(), self.arg())
```

```python
        # return "({: .3f}, {: .3f})".format(self.x, self.y)

    def __repr__(self):
        return "Vector2({0},␣{1})\n\t.norm␣=␣{2}\n\t.arg␣=␣{3}".format(self.x, self.y,
            ↪ self.norm(), self.arg())

    def norm(self):
        """Return the norm of the vector."""
        return math.sqrt(pow(self.x, 2) + pow(self.y, 2))

    def arg(self):
        """Return the angle of the vector."""
        return math.degrees(math.atan2(self.y, self.x))

    def set_mag(self, value):
        """Set vector's magnitude without changing direction."""
        if self.norm() == 0:
            logging.warning('Trying␣to␣set␣magnitude␣for␣a␣null-vector!␣Angle␣will␣be␣set
                ↪ ␣to␣0!')
            self.x = 1
            self.y = 0
        else:
            self.normalize()
        self.x *= value
        self.y *= value

    def set_angle(self, value):
        """Set vector's direction without changing magnitude."""
        if self.norm() == 0:
            logging.warning('Trying␣to␣set␣angle␣for␣a␣null-vector!␣Magnitude␣will␣be␣set
                ↪ ␣to␣1!')
            self.x = 1
            self.y = 0
        delta = angle_diff(self.arg(), value)
        self.rotate(delta)

    def rotate(self, value):
        """Rotate vector by degrees specified in value."""
        value = math.radians(value)
        self.x, self.y = math.cos(value) * self.x - math.sin(value) * self.y, \
                         math.sin(value) * self.x + math.cos(value) * self.y

    def normalize(self, ret=False):
        """Normalize the vector."""
        d = self.norm()
        if d:
            if not ret:
                self.x /= d
                self.y /= d
            else:
                return Vector2(self.x / d, self.y / d)

    def limit(self, value):
        """Limit vector's maximum magnitude to given value."""
        if self.norm() > value:
            self.set_mag(value)
```

```
    def limit_lower(self, value):
        """Limit vector's minimum magnitude to given value."""
        if self.norm() < value:
            self.set_mag(value)

    def constrain(self, old_value, max_value):
        """Limit vector's change of direction to max_value from old_value."""
        desired_value = self.arg()
        delta = angle_diff(old_value, desired_value)
        if abs(delta) > max_value:
            value = angle_diff(desired_value, old_value + math.copysign(max_value, delta)
                ↪ )
            self.rotate(value)


def angle_diff(from_angle, to_angle):
    diff = (to_angle - from_angle) % 360
    if diff >= 180:
        diff -= 360
    return diff


def pose_dist(pose1, pose2):
    """Return Euclidean distance between two ROS poses."""
    x1 = pose1.position.x
    y1 = pose1.position.y
    x2 = pose2.position.x
    y2 = pose2.position.y

    return math.sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2))


class MarkerSet(object):
    """
    Convenience class for handling Rviz markers.

    Markers are used to visualize each of the Reynolds' rules component in Rviz.
    Markers are set to arrows to represent force and velocity vectors.
    """
    def __init__(self):
        """Initialize class and set common marker properties."""
        self.visualization = MarkerArray()

        # Make sure these keys are the same as the ones in `boids.py`
        #keys = ['alignment', 'cohesion', 'separation', 'avoid', 'leader', 'acceleration
            ↪ ', 'velocity', 'estimated']
        keys = ['alignment', 'cohesion', 'separation', 'leader']
        self.markers = dict.fromkeys(keys)

        marker_id = 0
        for key in keys:
            self.markers[key] = Marker()
            self.markers[key].header.frame_id = rospy.get_namespace() + 'base_link'
            self.markers[key].header.stamp = rospy.get_rostime()
            self.markers[key].ns = rospy.get_namespace().split('/')[1]
```

```
            self.markers[key].id = marker_id
            self.markers[key].type = Marker.ARROW
            self.markers[key].action = Marker.ADD
            self.markers[key].pose = Pose()
            self.markers[key].pose.position.z = 0.036 # Sphero radius
            self.markers[key].lifetime = rospy.Duration(0)
            self.markers[key].frame_locked = True
            marker_id += 1

        # Set colors of each marker
        self.markers['alignment'].color = ColorRGBA(0, 0, 1, 1) # blue
        self.markers['cohesion'].color = ColorRGBA(0, 1, 0, 1) # green
        self.markers['separation'].color = ColorRGBA(1, 0, 0, 1) # red
        #self.markers['avoid'].color = ColorRGBA(1, 1, 0, 1) # yellow
        self.markers['leader'].color = ColorRGBA(0, 1, 1, 1) # light blue
        #self.markers['acceleration'].color = ColorRGBA(0, 0, 0, 1) # black
        #self.markers['velocity'].color = ColorRGBA(1, 1, 1, 1) # white
        #self.markers['estimated'].color = ColorRGBA(1, 0.55, 0, 1) # orange

    def update_data(self, values):
        """
        Set scale and direction of markers.

        Args:
            values (dict): Holds norm and arg data for each component
        """
        if values is not None:
            for key in self.markers.keys():
                data = values[key]
                angle = Quaternion(*quaternion_from_euler(0, 0, math.radians(data.arg())))
                scale = Vector3(data.norm(), 0.02, 0.02)

                self.markers[key].header.stamp = rospy.get_rostime()
                self.markers[key].pose.orientation = angle
                self.markers[key].scale = scale

            self.visualization.markers = self.markers.values()
        return self.visualization

    def get_heading(self, values):
        if values is not None:
            angle=0
            iter = 0
            for key in self.markers.keys():
                data = values[key]
                angle+= math.radians(data.arg())
                iter+=1
            return angle/iter
        return 0
```

J. setup_sim.launch

```
<launch>
    <!-- Set arguments. Change values in this section to control various parameters of
        ↪ execution. -->
```

```
<!-- There are some parameters that can be changed in other launch files but you should generally leave them as they are -->
<!-- ********** START OF SECTION ********** -->
<arg name="num_of_robots" default="2"/> <!-- Number of robots used. -->
<arg name="map_name" default="empty_10x10"/> <!-- Name of the map used. -->
<arg name="ctrl_loop_freq" default="10"/> <!-- Frequency used by Reynolds rules. -->
<arg name="data_stream_freq" default="10"/> <!-- Position streaming frequency, used by Kalman filter. -->
<arg name="debug_boids" default="false"/> <!-- Enable debugging for Reynolds controller node. -->
<arg name="debug_kalman" default="false"/> <!-- Enable debugging for Kalman filter node. -->
<arg name="use_kalman" default="false"/> <!-- Use either estimated data from Kalman filter or true data from simulator. -->
<arg name="wait_time" default="0"/> <!-- During first X seconds of execution, no velocity commands are sent to robots. -->
<arg name="start_time" default="2"/> <!-- During first X seconds after "wait_time", inital velocity commands are sent to robots. -->
<arg name="enable_leader_following" default="true"/> <!-- Enable Leader Following Behaviour-->
<!-- *********** END OF SECTION *********** -->

<arg name="map_world" default="$(find boids_ros)/resources/sim/$(arg map_name)_$(arg num_of_robots).world"/>
<arg name="map_yaml" default="$(find boids_ros)/resources/maps/$(arg map_name).yaml"/>

<!-- Set arguments as ros parameter so all nodes can access them. -->
<param name="num_of_robots" type="int" value="$(arg num_of_robots)"/>
<param name="ctrl_loop_freq" type="int" value="$(arg ctrl_loop_freq)"/>
<param name="data_stream_freq" type="int" value="$(arg data_stream_freq)"/>
<param name="debug_boids" type="boolean" value="$(arg debug_boids)"/>
<param name="debug_kalman" type="boolean" value="$(arg debug_kalman)"/>
<param name="use_kalman" type="boolean" value="$(arg use_kalman)"/>
<param name="wait_time" type="double" value="$(arg wait_time)"/>
<param name="start_time" type="double" value="$(arg start_time)"/>
<param name="run_type" type="string" value="sim"/>
<!-- Alperen-->
<param name="enable_leader_following" type="boolean" value="$(arg enable_leader_following)"/>


<!-- Start map server. -->
<node pkg="map_server" type="map_server" args="$(arg map_yaml)" name="map_server"/>

<!-- Start Stage simulator. -->
<node pkg="stage_ros" type="stageros" name="simulator" args="$(arg map_world)"/>

<!-- Start rqt GUI and dynamic reconfigure node. -->
<node pkg="rqt_gui" type="rqt_gui" name="rqt_gui"/>
<node pkg="boids_ros" type="dynamic_reconfigure_node.py" name="dyn_reconf" output="screen"/>

<!-- Start simulation_tf node: provide tf transforms for simulation. -->
<node pkg="boids_ros" type="simulation_tf.py" name="tf_server"/>
```

```
    <!-- Start rviz. -->
    <param name="robot_description" textfile="$(find␣boids_ros)/resources/simple_ball.
        ↪ urdf"/>
    <node pkg="rviz" type="rviz" name="rviz" args="-d␣$(find␣boids_ros)/launch/sim/
        ↪ sphero_sim.rviz"/>


</launch>
```

K. reynolds_sim.launch

```
<launch>
      <!-- Config file with initial velocities for each robot. -->
      <arg name="filename" default="$(find␣boids_ros)/cfg/sphero_init_vel.cfg"/>

      <!-- Start Reynolds controller nodes launcher. -->
      <node pkg="boids_ros" type="reynolds_launch.sh" name="reynolds_launcher" args="$
         ↪ (arg␣filename)␣robot" output="screen"/>

      <!-- Start nearest_search node: search for other robots in range. -->
      <node pkg="boids_ros" type="nearest_search.py" name="search" output="screen">
            <param name="robot_name" type="string" value="robot"/>
      </node>

      <!-- Move the leader via recorded rosbag file -->
      <!-- <arg name="rosbag_args" default='$(find␣boids_ros)/bagfiles/move_leader_0
         ↪ .5-1.bag'/> -->
      <!-- <node pkg="rosbag" type="play" name="rosbag_move_leader" args="$(arg␣
         ↪ rosbag_args)" output="screen"/> -->


      <arg name="node_start_delay" default="1.8" />
      <!-- Move leader -->
      <node pkg="boids_ros" type="leader_controller.py" launch-prefix="bash␣-c␣'sleep␣
         ↪ $(arg␣node_start_delay);␣$0␣$@'␣" name="leader_controller" output="screen"
         ↪ />

      <!-- Record Bagfile for data_analyzer -->
      <arg name="bagname"/>
      <node pkg="rosbag" type="record" name="rosbag_record" args='-O␣$(find␣boids_ros)
         ↪ /bagfiles/$(arg␣bagname).bag␣-e␣"(.*)/odom"␣"(.*)/cmd_vel"␣␣'/>

<!-- Record a bag for debug purposes -->
      <!-- <arg name="rosbag_args" default='-O␣$(find␣boids_ros)/bagfiles/sim_test.bag
         ↪ ␣-e␣"(.*)/odom"␣'/> -->
      <!-- <arg name="rosbag_args" default='-O␣$(find␣boids_ros)/bagfiles/kalman_test.
         ↪ bag␣/robot_0/odom␣/robot_0/debug_est'/> -->
      <!-- <node pkg="rosbag" type="record" name="rosbag_record" args="$(arg␣
         ↪ rosbag_args)" output="screen"/> -->
</launch>
```

CODE WRITTNE ONLY BY Marko Križmančić:

L. simulation_tf.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Broadcast tf data during simulation.

tf data is produced from position of each robot received on Odometry messages.
It is used to visualize simulated robots in Rviz.
"""

import rospy
import tf2_ros
import geometry_msgs.msg
from nav_msgs.msg import Odometry


def callback(msg):
   global tfBuffer
   try:
      tf = geometry_msgs.msg.TransformStamped()
      tf.child_frame_id = msg.header.frame_id
      tf.header.frame_id = "map"
      tf.header.stamp = msg.header.stamp

      tf.transform.translation.x = 0
      tf.transform.translation.y = 0
      tf.transform.translation.z = 0

      tf.transform.rotation.x = 0
      tf.transform.rotation.y = 0
      tf.transform.rotation.z = 0
      tf.transform.rotation.w = 1

      broadcaster.sendTransform(tf)

   except BaseException as exc:
      print exc
      tfBuffer.clear()
      return


if __name__ == '__main__':
   rospy.init_node('simulation_tf')
   tfBuffer = tf2_ros.Buffer()
   listener = tf2_ros.TransformListener(tfBuffer)
   broadcaster = tf2_ros.TransformBroadcaster()

   num_of_robots = rospy.get_param("/num_of_robots")
   [rospy.Subscriber("/robot_{}/odom".format(i), Odometry, callback) for i in range(
      ↪ num_of_robots)]

   rospy.spin()
```