ROB-GY.6333 Networked Robotics Systems, Cooperative Control and Swarming
Final Project Report

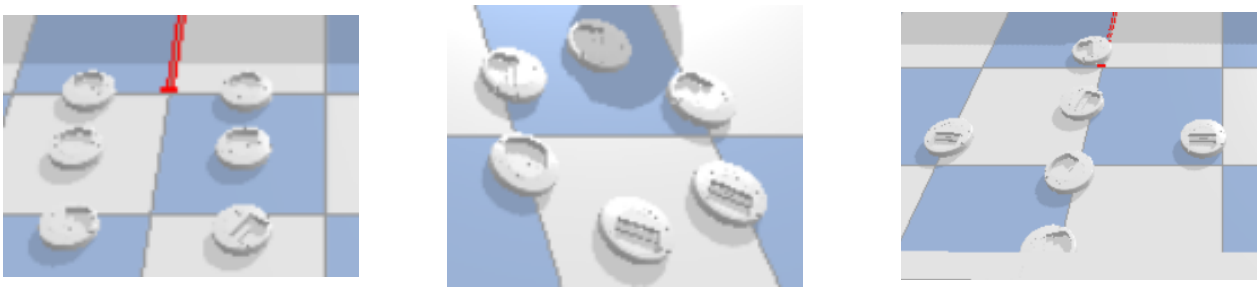
Irving Fang

Fall 2021

Contents

1 Results

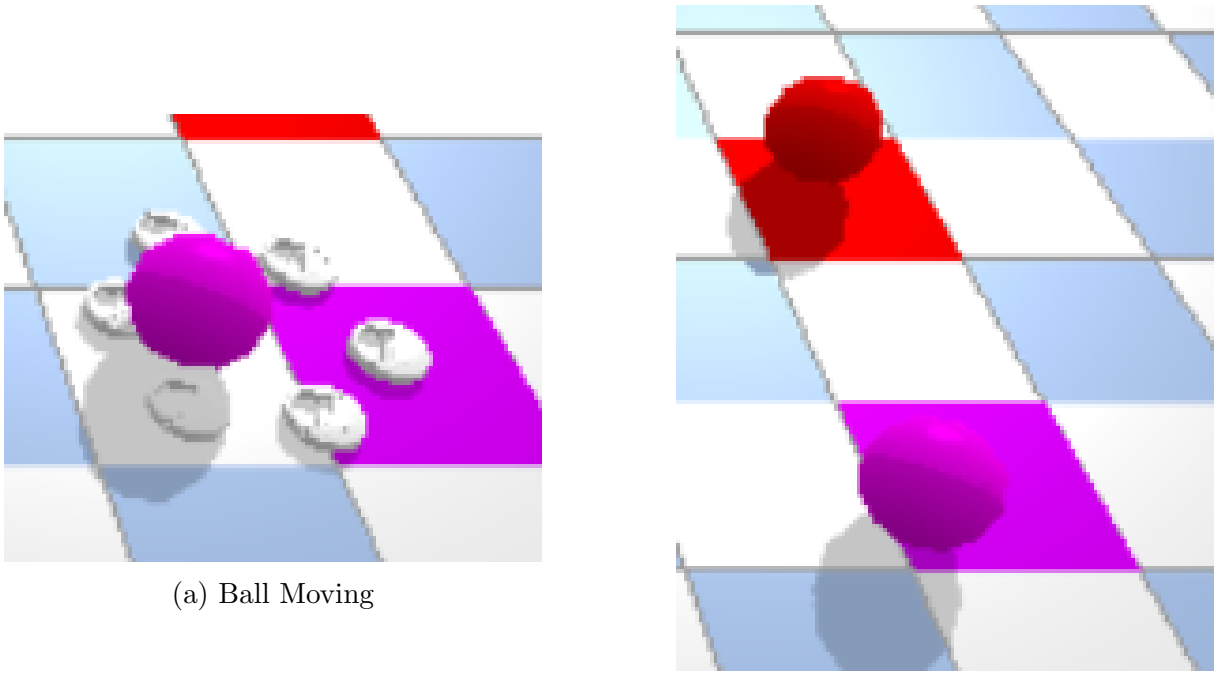
Here I briefly present the results of the final project. For a more detailed presentation, please refer to the video I uploaded with the report and code. Please read the Miscellaneous section (section 3) before reading the report and watching video if possible! It explains some caveats that doesn't fit in the report.



(a) Square Formation (b) Circle Formation (c) Diamond Formation

Figure 1: Formations for Different Tasks

Figure 1 showcases the three formations we are asked to achieve.



(a) Ball Moving (b) Balls Final Position

Figure 2: Ball Manipulation

Figure 2 showcases the collective ball manipulation. We can see the caging movement and that the two balls ends up at the designated location.

2 Theory and Implementation

Here I will introduce the theory and implementation to achieve the five tasks. Few general rules:

1. I will introduce every parameter when they first appears. Parameters with same name in different methods serve the same purpose.
2. All algorithms are decentralized algorithms.
3. All computations involve robot states are doing with x and y coordinates **separately** and **independently**.
4. No collision between robots and between robots and wall as I implemented potential field for obstacle avoidance.

2.1 Custom Fields

Here are some of the fields I defined to facilitate the decentralized algorithms.

1. `self.delta`: The maximum distance of inter robot communication. Used mainly for formation control algorithm.
2. `self.t`: Timer variable. Used to keep track of how many iterations a specific decentralized algorithm has been running so the robot can switch to the next decentralized algorithm when one task is finished.
3. `self.desired_coor_dict`: Dictionary that holds the reference coordinates for formation control. Keys are string such as "square". Values are the reference coordinate corresponding to the string.
4. `self.is_surround`: Flag to check if the ball is surrounded by all the robots. Used for caging balls. More details below.
5. `self.in_shape`: Flag to check if the formation is roughly in shape. Used when forming the diamond shape. More details below.

2.2 Task 1: Make a square formation in the room

This task is achieved with weighted formation control algorithm, through methods `formation_controller()` and `edge_tension_dynamics()` in `Robot.py`. Basically, we will utilize the edge tension function to enhance our consensus algorithm, so that we can obey the 2m communication limits and have a decentralized algorithm.

1. `formation_controller(self, formation, curr_id, curr_pos, messages)`:
   (a) `formation`: string that corresponds to the formation I want the swarm of robots to make.
   `curr_id`: the current id of this robot.
   `curr_pos`: the current coordinates of this robot.
   `messages`: all messages this robot receives, which contains coordinates of all neighbors.
   (b) It reads the `self.desired_coor_dict` to acquire the matrix representing desired coordinates, according to the `formation` I pass in, which is `'square'` in this task.
   (c) For every neighbor of the current robot, it calculates the desired distance between the neighbor and the current robot using the desired coordinates. Then it passes the desired distance, current robot's coordinates and neighbor robot's coordinates to `edge_tension_dynamics()` to acquire `dx, dy` for a single neighbor.

(d) After acquire `dx, dy` for every neighbor, it sums up all the `dx, dy` and returns the final `dx, dy` that can be used to calculate the wheel velocities using the provided code.

(e) Because sometimes the velocities calculated will be too large for PyBullet simulation, I clipped the `dx, dy`. The upper and lower limit is set through trial-and-error to 50.

2. `edge_tension_dynamics(self, d_ij, x_i, x_j)`:

(a) `d_ij`: desired distance between two robots.
`x_i`: coordinate of the current robot.
`x_j`: coordinate of the neighbor robot.

(b) This methods implement the control law introduced in Lecture 8:

$$-\frac{2(\Delta - ||d_{ij}||) - ||l_{ij}(t) - d_{ij}||}{(\Delta - ||d_{ij}|| - ||l_{ij}(t) - d_{ij}||)^2}(x_i(t) - x_j(t) - d_{ij}) \tag{1}$$

Here, Δ is `self.delta`, which is the communication distance limit. d_{ij} is the desired distance between robot i and robot j. l_{ij} is the current distance between robot i and robot j at time t. $x_i(t)$ is the current coordinates of robot i at time t.

(c) Note that in this method, I simply calculate the dx, dy for a **single neighbor**. The complete control law requires summing all the dx, dy from all the neighbors, which is done in `formation_controller()`

(d) Though not the focus of this specific method, the theory behind the control law is also introduced in Lecture 8. First we have edge tension function:

$$V_{ij}(\delta, x) = \begin{cases} \dfrac{||l_{ij}(x)||^2}{\delta - ||l_{ij}(x)||} & \text{if } v_i, v_j \in E \\ 0 & \text{if otherwise} \end{cases} \tag{2}$$

And we have the control law:

$$u_i = -\sum_{j \in N_i} \frac{2\delta - ||l_{ij}||}{(\delta - ||l_{ij}||)^2}(x_i - x_j) \tag{3}$$

Take the derivative of tension function and combine these two to have a formation control algorithm, and we have the decentralized control law above.

3. The desired coordinates I used for `'square'` is $[[-0.5, -1], [-0.5, 0], [0, -1], [0, 0], [0.5, -1], [0.5, 0]]$. It's treating the robot at row 2 col 2 as the origin.

2.3 Task 2: Get all the robot out of the room and form a circle

This task is achieved with potential field algorithm, through methods `potential_field()`, `follow_leader()` and `surround_ball()`. Basically, I set a potential field outside of the room using `follow_leader()` and add potential field to the two walls that's closed to the exit so the robots can get out the room without colliding with the room. Finally, I use `surround_ball()` to form a circle outside.

1. `potential_field(self, r_i, r_j, d_0=1.5, alpha=5)`:

(a) `r_i`: current object's current coordinates.
`r_j`: another object's current coordinates.
`d_0`: the distance parameter used in potential field.
`alpha`: the α parameter used in potential field.

(b) The potential field between objects is defined as:

$$V_I = \begin{cases} \alpha_I(\log(r_{ij}) + \frac{d_0}{r_{ij}}) & \text{if } 0 < r_{ij} < d_1 \\ \alpha_I(\log(d_1) + \frac{d_0}{d_1}) & \text{if } r_{ij} \geq d_1 \end{cases} \tag{4}$$

We take the derivative of V_I with respect to r_ij, and we have the the force the current object experience is the sum of:

$$V_I' = \begin{cases} \alpha_I(\frac{1}{r_{ij}} - \frac{d_0}{r_{ij}^2}) & \text{if } 0 < r_{ij} < d_1 \\ 0 & \text{if } r_{ij} \geq d_1 \end{cases} \tag{5}$$

over all the interacting objects. The `potential_field()` methods then implements the above potential field. So basically, when a location has a potential field, object too close to the field will be repulsed, and object reasonably far away from it will be attracted.

2. `follow_leader(self, leader_pos, curr_pos, messages, distance)`

   (a) `leader_pos`: The coordinates outside the room that our robots will get to.
   `distance`: The distance between each robot.

   (b) First it sets a potential field `potential_field()` at a point outside the room using `leader_pos`, so the robots will be attracted to that point.

   (c) Then, it adds potential fields to the two walls that may become obstacles using helper function `wall.room_potential()`, which I defined in `wall.py`. Omit its details due to limited pages.

   (d) Finally, it add potential field for each robot using `potential_field()`, so that robots keep `distance` away from each other when getting outside the room.

3. `surround_ball(self, ball_pos, curr_pos, obstacle_pos, messages, radius)`

   (a) `ball_pos`: coordinates of the ball that robot will circle around
   `obstacle_pos`: coordinates of the obstacle the robots will avoid.
   `radius`: the radius of the circle the robots form.

   (b) Originally, I planned to use `formation_controller()` for form a circle, but calculating reasonable reference coordinates for a circle formation is a bit tricky. A serendipity is that a potential field can naturally make robots form a circle, so I ended up with this, as it will add a potential field at `ball_pos`

   (c) I set `ball_pos` as a coordinates outside the room. Set no obstacle, and the radius is a arbitrary radius, so the robot will form a circle at the `ball_pos`.

2.4 Task 3: Move the purple ball on the purple square

This task is achieved by setting a potential field at the ball with `surround_ball()` so robots will circle around the ball close enough to be able to move it, and then use `get_formation_pos()` and `push_ball()`, which implements formation control and target moving, to move to the target location. Finally, call `surround_ball()` again with bigger radius to release the ball.

1. `surround_ball(self, ball_pos, curr_pos, obstacle_pos, messages, radius)`

(a) We set `ball_pos` to the coordinates (2, 4) of the purple ball. `obstacle_pos` is `None` for now. `radius` is `0.7` so the ball will not slip through from the cage.

2. `get_formation_pos(self)`

   (a) This method calculates the current coordinates after the surrounding is finished and record it to the `self.desired_coor_dict` so when moving the ball, the robots will keep the current tight cage.

   (b) I set `self.is_surround` to `True` after the surrounding is finished, so the desired coordinates will be the tight cage, and the pushing can begin.

3. `push_ball(self, curr_id, curr_pos, desired_pos, target_pos, messages)`

   (a) `target_pos`: the destination tile coordinates.

   (b) This method basically is just `formation_controller()`, except I now also calculate the difference between the center of this circle formation at the current time and the `target_pos` (which is a point in the purple tile (2.6, 5.6). It's finetuned so the ball will not slip away after we release it), and add the difference to the `dx, dy`, so the robots will keep the caging formation while moving to the designation tile.

4. call `surround_ball()` again with the same parameters, except this time `radius` is `1.5`, so robots release the ball from caging by forming a large ball around it.

2.5 Task 4: Move the red ball on the red square

This task is achieved almost identical to Task 2.4, with some small caveats:

1. When calling `surround_ball()`, `obstacle_pos` is now (2.6, 5.6), so a potential field will be added to the purple tile, so robots will not collide with the purple ball.

2. When calling `push_ball()`, I call `wall.room_potential()` again so the pushing robots will not collide with the wall. `target_pos` is (0.3, 5.6). Again, finetuned so ball won't slip out the tile.

2.6 Task 5: Get back into the room and make a diamond formation

1. Call `surround_ball()` to go to a point close to the exit of the room. `obstacle_pos` is now (0.3, 5.6) to avoid colliding with the red ball.

2. Call `follow_leader()` to get back inside the room without colliding with the wall.

3. Call `formation_controller()` to form a diamond. Note that for diamond shape, I add potential field to each robot, unlike when forming square, so robots will not collide with each other. After a few time step when the robots are roughly in their expected position but still moving very slowly to get to the exact position, I turn `self.in_shape` to `True` so the potential field is turned off, as formation control with potential field is very slow.

4. The desired coordinates I used for `'diamond'` is $[[0, 0], [2/3, 0], [4/3, 0], [2, 0], [1, 1], [1, -1]]$.

3 Miscellaneous

1. Because of the viewing angle of PyBullet, which I haven't find a way to modify, Figure 1a looks a bit like a rectangular. However, if the viewing angle is vertical we would see that it's actually a square.
2. In Figure 2b, the purple ball is a bit tilted. This is because the rolling friction of this simulation environment is very small, so it's hard to keep the ball 100% static when it reaches the destination. However, the contact point is still entirely on the purple tile, so it should be okay.
3. I deliberately add a gap between each algorithm so the robots will stay still for a bit, so you can see the formation and other effect more clearly. It's not a bug.
4. Currently, switching between different decentralized algorithm is done by manually setting iteration number of each algorithm. The specific number is acquired by trial-and-error, which is not ideal. I implemented a method to detect if an algorithm is finished by checking if the robots are relatively static, but it performs not as ideal as I expect. Since it's not the emphasis of this project, I didn't submit the version with that function.
5. I didn't explained too much about the potential field of walls. Please read `wall.py` for more details. It's inspired by this implementation of Github user BolunDai0216 about wall potential field.
6. A more honest approach to pushing ball is that we always add obstacle avoidance for the other ball that's not being moved and walls. However, the specific lay out in this project allows us to move each ball without worrying about colliding with the other. And when moving the purple ball we don't need to consider the walls. So the current way of doing it will only work for this project layout.
7. The diamond formation is a bit slow at the moment due to potential field. Please be patient.
8. There is a very very small chance that the simulation will fail. If that unfortunately happens, which is very unlikely, please rerun the program! One mysterious reason it fails is that if I switch np.power(x, 2) with np.square(x), which are equivalent, it will fail.