CS 286 - Homework 1, Nerd-Herd Part B

Adolfo Roquero, Leticia Schettino, Kobi Greene

February 2022

1 Write up

bot_sense

The ***bot_sense*** function is not used in any of part A) but is extremely important for all of the functions implemented in part b since the decentralized communication and control approach makes use of local communication for robots within a sensing range. The ***bot_sense*** function takes the sensing radius, *sense_r*, as a parameter and gets the neighbors within *sense_r* of each neighbor. In our implementation of ***bot_sense***, we iterate over all of the robots in the environment then, for each robot, iterate over a smaller portion of the grid defined by the coordinate of the robot in the grid $\pm$ *sense_r* in each coordinate direction and add all of the robots in the grid to the list of neighbors for that robot. Since a robot cannot be a neighbor of itself, we remove the current robot from the list before continuing the iteration.

update_flock

The ***update_flock*** generates the flocks at each timestep based on the position of each robots and robots within their neighborhood. While there are many possible ways to generate a flock, we wanted our implementation to allow for all of the robots to belong to the same flock in part a (since we were exploring with centralized communication and control) and for the flocks to be generated based on the sensing radius in part b (since we were exploring decentralized communication and control).
The way in which we achieved this was to have a flock being represented by a set of robots and initializing the flock property of the environment to a list containing the set of all robots. Then, since in part a there is no changes to flock membership because every robot has global information, we simply have no calls to ***update_flock*** or ***bot_sense***. For part b, however, when the concept of sensing radius and neighbours are introduces as part of decentralized communication and control, we make use of the function ***bot_sense*** and once we know what are the neighbours of each robot, we generate flocks using a greedy algorithm, which ensures that a robot can only be part of a single flock. The greedy algorithm proceeds by erasing any existing flocks and starting with an

empty list of flocks. Then, it proceeds by iterating over all of the robots in the environment and checking whether they belong to a flock already or not, if they do not, then they are added to a new flock, as long with all of their neighbours that do not already belong to another flock. There are other ways in which the algorithm could be implemented, such as joining two flocks everytime a neighbour is about to be added but already belongs to another flock however, to stick to a more decentralized approach where robots only communicate with robots in their flock and the flock is determined by sensing radius, we opted for the greedy implementation.

safe_wander_sense

The ***safe_wander_sense*** function takes a parameter, *flock*, which determines whether the bots in a flock should wander in the same random direction or whether each bots move in their own random direction. This is the same behavior we implemented in part a, for the ***safe_wander*** function and therefore, ***safe_wander_sense*** simply makes a call to the class function ***safe_wander***.

aggregate_sense

The intended behavior of the ***aggregate_sense*** method is to have all robots in the environment in the same position in the grid such that the function only terminates once this convergence is achieved. Given that in part b we are working with decentralized communication and the robots can only communicate to robots in their sensing radius (i.e. their neighbours), our implementation of ***aggregate_sense*** first ensures that all robots belonging to a same flock aggregate in the centroid of that flock. Then, if the robots are not all in the same position after this step, the flocks will all wander around. Once the flocks have wondered, the function calls for an ***update_flock***, which will check whether there are new, bigger flocks that should be formed by effectively joining two or more existing flocks. Once the flocks have been updated, the first steps mentioned of iterating through the flocks and having the robots in each flock aggregate in their centroid (by a call to ***aggregate***) are repeated and the whole procedure is repeated again until every robot is in the same position in the grid.

disperse_sense

The desired behavior of the ***disperse_sense***. function is to move all robots away from their respective flock's centroid. Sicne this is the same behaviour implemented in part A) for the ***disperse*** function, the ***disperse_sense*** function simply makes a call to the class function ***disperse***. The implementation of ***disperse*** is described in more detail in part a but it is worth noting that every call to ***disperse*** will have the robots moving for 3 steps and if any robot is currently in the centroid of their flock, then they will move towards a random direction for 3 steps.

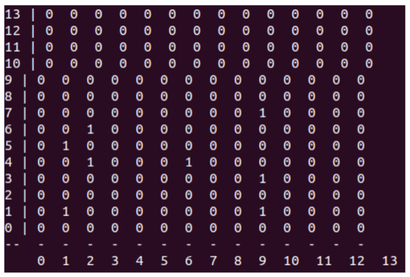
flock_sense

The ***flock_sense*** function implementation was outlined in the distribution code and makes calls to ***aggregate_sense***, ***safe_wander_sense*** and ***disperse_sense***. The intended behavior is to have all the robots converge to a single position in the grid, then wander randomly as a flock for t steps then disperse from the point they are aggregated into. It is very similar to the ***flock*** function, however it has the extra *sense_r* parameter which is passed into ***aggregate_sense*** to ensure that the robots follow a decentralized communication and control and that they can only communicate with robots within their flock when aggregating.

2 Visualizations

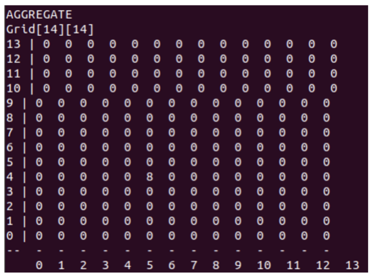
Given the function ***flock_sense*** calls all of the functions included above, it is sufficient to make a call to ***flock_sense*** and include the resulting grids to demonstrate how each of the functions work. The function call that generated these images was a call to ***flock_sense*(3)** on an environment with 8 robots initially randomly scattered across the grid.

The first call within the *flock_sense* function is a call to *aggregate_sense* which will perform the logic described above until all robots in the environment are in the same position:



```
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
7 | 0  0  0  0  0  0  0  0  0  1  0  0  0  0
6 | 0  0  1  0  0  0  0  0  0  0  0  0  0  0
5 | 0  1  0  0  0  0  0  0  0  0  0  0  0  0
4 | 0  0  1  0  0  0  1  0  0  0  0  0  0  0
3 | 0  0  0  0  0  0  0  0  0  1  0  0  0  0
2 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  1  0  0  0  0  0  0  0  1  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

Figure 1 - initial grid with 8 robots

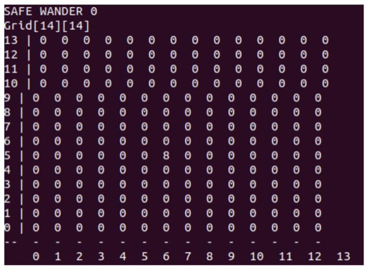


```
AGGREGATE
Grid[14][14]
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
7 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
6 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
5 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
4 | 0  0  0  0  0  8  0  0  0  0  0  0  0  0
3 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
2 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

Figure 2 - updated grid after aggregate_sense call, all 8 robots at position (5, 4)

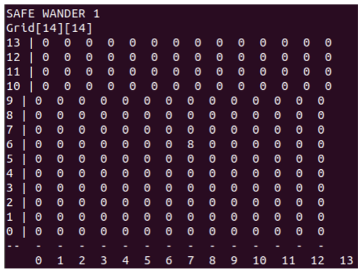
The images above show the initial starting position of the 8 robots and the final position where they aggregated to. The intermediary steps of how the robots will move towards the centers of their flocks then each flock will wander until bigger flocks are formed are not shown in these images however, the behavior can be visualized by the turtlesim plots. After a single call to *aggregate_sense*, all robots are in the same position in the grid.

Once all robots have aggregated in the same grid position, the function has the flock safe wander for t steps (default t = 5 shown here). The figures below show the behavior of the *safe_wander_sense* function:



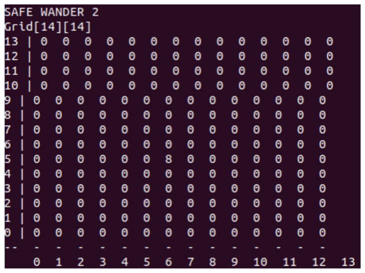
```
SAFE WANDER 0
Grid[14][14]
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
7 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
6 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
5 | 0  0  0  0  0  0  8  0  0  0  0  0  0  0
4 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
3 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
2 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

Figure 3a - all robots at position (6, 5)



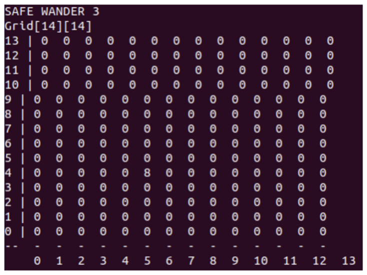
```
SAFE WANDER 1
Grid[14][14]
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
7 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
6 | 0  0  0  0  0  0  0  8  0  0  0  0  0  0
5 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
4 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
3 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
2 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

Figure 3b - all robots at position (7, 6)



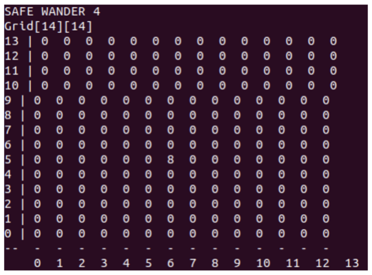
```
SAFE WANDER 2
Grid[14][14]
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
7 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
6 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
5 | 0  0  0  0  0  0  8  0  0  0  0  0  0  0
4 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
3 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
2 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

Figure 3c - all robots at position (6, 5)



```
SAFE WANDER 3
Grid[14][14]
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
7 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
6 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
5 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
4 | 0  0  0  0  0  8  0  0  0  0  0  0  0  0
3 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
2 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

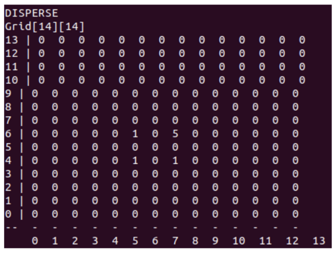
Figure 3d - all robots at position (5, 4)



```
SAFE WANDER 4
Grid[14][14]
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
7 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
6 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
5 | 0  0  0  0  0  0  8  0  0  0  0  0  0  0
4 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
3 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
2 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

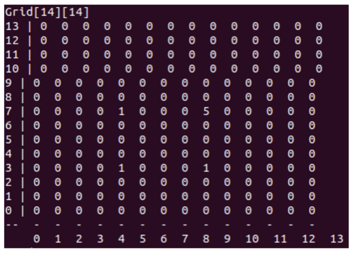
Figure 3e - all robots at position (6, 5)

After the flock has wandered, the function has them disperse beyond the aggregation centroid. The figures below show the behavior of the *disperse_sense* function:



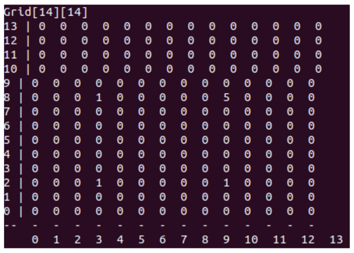
```
DISPERSE
Grid[14][14]
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
7 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
6 | 0  0  0  0  0  1  0  5  0  0  0  0  0  0
5 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
4 | 0  0  0  0  0  1  0  1  0  0  0  0  0  0
3 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
2 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

Figure 4a - grid after call to disperse_sense (step 1)



```
Grid[14][14]
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
7 | 0  0  0  0  1  0  0  0  5  0  0  0  0  0
6 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
5 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
4 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
3 | 0  0  0  0  1  0  0  0  1  0  0  0  0  0
2 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

Figure 4b - grid after call to disperse_sense (step 2)



```
Grid[14][14]
13 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
12 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
11 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
10 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
9 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
8 | 0  0  0  1  0  0  0  0  0  5  0  0  0  0
7 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
6 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
5 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
4 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
3 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
2 | 0  0  0  1  0  0  0  0  0  1  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0
--  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    0  1  2  3  4  5  6  7  8  9  10  11  12  13
```

Figure 4c -grid after call to disperse_sense (step 3)

The image above highlights the implementation described in part a where the call to disperse will have robots moving in the direction that is opposite to the vector between the robot and the centroid. If the robot is at the centroid (which is the case for all the robots after they have aggregated) then the function will generate random directions for each robot to move towards. Due to the randomness in the disperse function, some robots might end up dispersing in the same direction, which is the case with some of the robots in this image. Each call to disperse will have the robots move 3 steps towards the direction opposite to the centroid (or random direction) and the images show what happens at each step.