

Distributed Intelligence in Autonomous Robotics

Assignment #2

Out: Tuesday, January 28, 2003

Due: Thursday, February 6, 2003

Aggregation and Dispersion

The purpose of this assignment is to create the building blocks for multi-robot flocking behaviors. We will start with creating the aggregation and dispersion behaviors, following after the approach of Mataric (assigned reading #4). Since the Nomad200 simulator cannot handle a huge number of robots, we will apply this software development to 6 robots.

From Assignment #1, you learned how to run multiple robots under the Nomad200 simulator. For this exercise, you may reuse any of the code from the example exercises (in the Nomad200/client/Examples directory) that is appropriate and useful. For all of this exercise, you will be generating code that will be run on each robot individually (just like the leader/follower example from the Multiple_robots example code). That is, the control will be distributed, with each robot determining its own motions based on its own local situation. Remember from Assignment #1 that you can automatically start up 6 robots in the Nserver simulator by modifying the world.setup file to have a list of 6 robot.setup files using the “setup_files=” parameter.

For this assignment, you should turn in the output requested below in the section entitled “TURN IN THE FOLLOWING” on page 4.

NOTE: In this exercise, there is no explicit code for preventing collisions between robots. You may need to incorporate extra conditions to handle this, such as ensuring that the initial robot placements are at least some minimum distance apart from each other, or making a robot move away from another robot if they are closer than some small distance from each other. Experiment first without using these sort of conditions, but if you aren't able to make your code work otherwise, then add these special cases and discuss what you did in Part D of what you turn in (listed on page 4 below).

1. User input

In this exercise, there are several parameters that will be supplied by the user when invoking the program. Your code should handle these parameters using argc and argv. These user-defined parameters are summarized in the following table:

Parameter	Definition
run_type	Indicates whether this run is to be a “robot team disperse” run or a “robot team aggregate” run. Types should be either ‘d’ or ‘a’.
initial_radius	Radius of circle within which robots will be initially placed, in units of tenths of inches.
d_sense	“Sensing range” of a robot, in units of tenths of inches. The centroids will be calculated based upon this sensing range.
distance	If this run is of type “robot team disperse”, then distance = d_disperse, which is the minimum inter-robot distance for dispersion, in units of tenths of inches. If this run is of type “robot team aggregate”, then distance = d_aggregate, which is the maximum inter-robot distance for aggregation, in units of tenths of inches.

Your code should enforce the following condition:

$$\text{distance} < d_sense$$

If the user inputs parameters that do not meet this condition, your code should print out an informative error message stating the problem, and then exit.

2. Initial robot distribution

Write code to generate the initial positions and orientations of the 6 robots according to the following instructions. The initial x,y positions of the robots should be randomly distributed within a circle centered at (0,0) and with radius "initial_radius". The initial_radius variable should be treated as a parameter to be passed into the program (using argc and argv), to enable the user to specify different values at runtime. The initial orientation of the robots should also be randomly generated between 0 and 360 degrees. Be sure your robots' positions and orientations are **uniformly** distributed within the defined circle. ALSO: be sure that the initial positions and orientations generated by your code will be different on each run of your program (i.e., you don't want the same positions and orientations for each run). One way to accomplish this initial distribution is to use a random number generator three times. First, generate the x position in the range (-initial_radius, +initial_radius). Second, depending upon the x position generated, calculate the y position so that the resulting x,y position remains within the defined circle of radius initial_radius (use standard geometry and the pythagorean theorem to calculate these values). And, third, use another random number invocation to generate the orientation between 0 and 360 degrees. Depending on the random number generator you use, you may need to provide a seed for starting the number generation. As a suggestion, you could make the seed be a function of the time and day so that it generates different random numbers on each invocation.

Use the "place_robot" Nomad200 command to place the 6 robots in these random positions and orientations at the beginning of the program. Note that the units that are used in the Nomad200 simulator are tenths of inches and tenths of degrees. So, "place_robot (100,100,900,900)" will place the robot at x = 10 inches, y = 10 inches, with orientation of 90 degrees.

3. Communicate robot positions

Since it is very difficult to detect robot positions using the Nomad200 sensors, we will have robots communicate their current positions and orientations to their teammates. Use the broadcast communications method described in Assignment #1 to have robots communicate their current positions and orientations periodically to their teammates. Example code for exchanging this information is in the leader.c and follower.c code in Nomad200/client/Examples/Multiple_robots. However, for this exercise, use the broadcast method instead of the point-to-point method for communication. Thus, rather than having the procedure call "talk_to_robot('P', '2', msg)", you would instead use "talk_to_robot('B', '0', msg)", so that each position/orientation message is communicated to all robots. The msg variable would include the robot's position and orientation, as illustrated in the "communicate" subroutine in the leader.c code. However, note that the leader.c communicate subroutine currently does not communicate the robot's current orientation, but instead its desired direction of motion. Your code for this exercise should communicate the robot's current orientation.

When each robot receives a position message from its teammates, it saves the robot's current position and orientation in an array for use in calculating the centroid.

4. Computing Centroid

Write a function “`calc_centroid`” that accepts a parameter “`centroid_distance`” from the calling subroutine and calculates the centroid of all neighboring robots that are within the distance `centroid_distance` from the current robot. In this case, the centroid is the average x,y position of the robots within this range. Note that the result of this calculation will be different for each robot, depending upon its own local environment. Be sure not to include robot positions in the calculation that are not within `centroid_distance` of the current robot.

5. Dispersion

Write a subroutine “`disperse`” that executes the disperse algorithm given in Mataric (page 9). As is given in this algorithm, the parameter “`d_disperse`” should be a parameter that the user can change from run to run. Your `calc_centroid` subroutine should be called with a parameter of `d_sense` for calculating the centroid. Be sure to stop the robot if it does not need to move away from the centroid. Your code for moving the robot away from the centroid will be similar to the `navigate/pilot/act` functions in the `leader.c` code in the `Nomad200/client/Examples/Multiple_robots` directory (except that you will be moving directly away from the calculated centroid instead of toward it).

6. Aggregation

Write a subroutine “`aggregate`” that executes the aggregate algorithm given in Mataric (page 11). As is given in this algorithm, the parameter “`d_aggregate`” should be a parameter that the user can change from run to run. Your `calc_centroid` subroutine should be called with a parameter of `d_sense` for calculating the centroid. Be sure to stop the robot if it does not need to move toward the centroid. Your code for moving the robot toward the centroid will be similar to the `navigate/pilot/act` functions in the `leader.c` code in the `Nomad200/client/Examples/Multiple_robots` directory.

7. Main routine

Write the main function, which accepts the user input, places the robots in their initial positions, and, depending upon whether the user has specified “`aggregate`” or “`disperse`”, calls the appropriate function (in an infinite loop) to cause the robots to execute the proper behavior. Your robots should eventually reach a steady state where no more motions are needed in order to meet the conditions for `aggregate` and `disperse`.

8. Evaluation of code convergence

To evaluate the time required for your code to reach a steady state, where no more robot motions occur, run your code 10 times for each of the `aggregate` and `disperse` run types. Your initial parameter settings should be as follows:

Parameter	Definition
<code>run_type</code>	‘d’ or ‘a’
<code>initial_radius</code>	1400
<code>d_sense</code>	1400
<code>distance</code>	If this run is of type “robot team disperse”, then <code>distance = d_disperse = 1000</code> If this run is of type “robot team aggregate”, then <code>distance = d_aggregate = 700</code>

For each run, calculate the clock time from the start of the run until the steady state is reached. Plot these values on two charts (one each for disperse and aggregate), with the x-axis giving the run number and the y-axis giving the calculated time. Note that the 10 times for each chart will likely be different from each other, since the robots are starting in different initial positions. Calculate and note the average time and standard deviation for each set of 10 runs. (Microsoft Excel provides easy utilities for creating this type of plot, although any graphics package you want to use is fine.)

9. Code documentation

Write documentation at the top of your code that includes the following information:

- Your name
- File name of your code
- “CS594 Assignment #2”
- A short description of what the code does
- A description of the compile command needed to recompile your code
- A description of the commands needed to execute your multi-robot system, clearly stating the complete series of commands that should be issued and the order in which they should be issued. Be sure to include the invocation of the Nserver and the communication server code as part of your instructions. If your code does not use the default Nserver port value of 7019, then your instructions should also state what port number you used.

Be sure to confirm the correctness of your compile command and your system execution instructions by stepping through the commands yourself to make sure they work in a new test directory. ***Points will be deducted from your assignment if your compilation or execution instructions are missing, incomplete, unclear, or incorrect, or if your code has compiler errors, or if files are missing.***

TURN IN THE FOLLOWING:

- A.** A screen dump showing the initial distribution of robots for one of your runs and a screen dump of the final convergent state of the robots after executing the “disperse” behavior. Be sure that your starting screen dump and the final screen dump are from the same run of your code and at the same resolution.
- B.** A screen dump showing the initial distribution of robots for one of your runs and a screen dump of the final convergent state of the robots after executing the “aggregate” behavior. Be sure that your starting screen dump and the final screen dump are from the same run of your code and at the same resolution.
- C.** Your two plots of data collected from Part 8 above, including your calculated average times and standard deviations. Your plots should have a title, and the axes should be clearly marked with labels and axis values.
- D.** A discussion of any issues you had to deal with in order to make your code work properly. For example, one of the issues you may have had to deal with was preventing collisions between robots.
- E.** A hardcopy of your code, fully documented according to Part 9 above. You do not have to provide hardcopies of any code that is provided to the class (e.g., communications_server.c)
- F.** Create a tar or zip file of your code, including all files needed to compile your code in an empty directory (e.g., Nclient.o, Nclient.h, inter_robot_communications., etc.). You do not have to include the communication_server.c code. ***Be sure to test the unpacking and recompilation of your code in an empty directory, to ensure that you have included all necessary files for code compilation.*** Email a hardcopy of your code package to parker@cs.utk.edu, naming your code “yourlastname-2.tar” or “yourlastname-2.zip”.