Networked Control and Multi-Agent Systems
Final Project Instructions

By Philip Twu and Magnus Egerstedt
Last Updated on August 23, 2013 by Greg Droge

Section 1: Mission Briefing

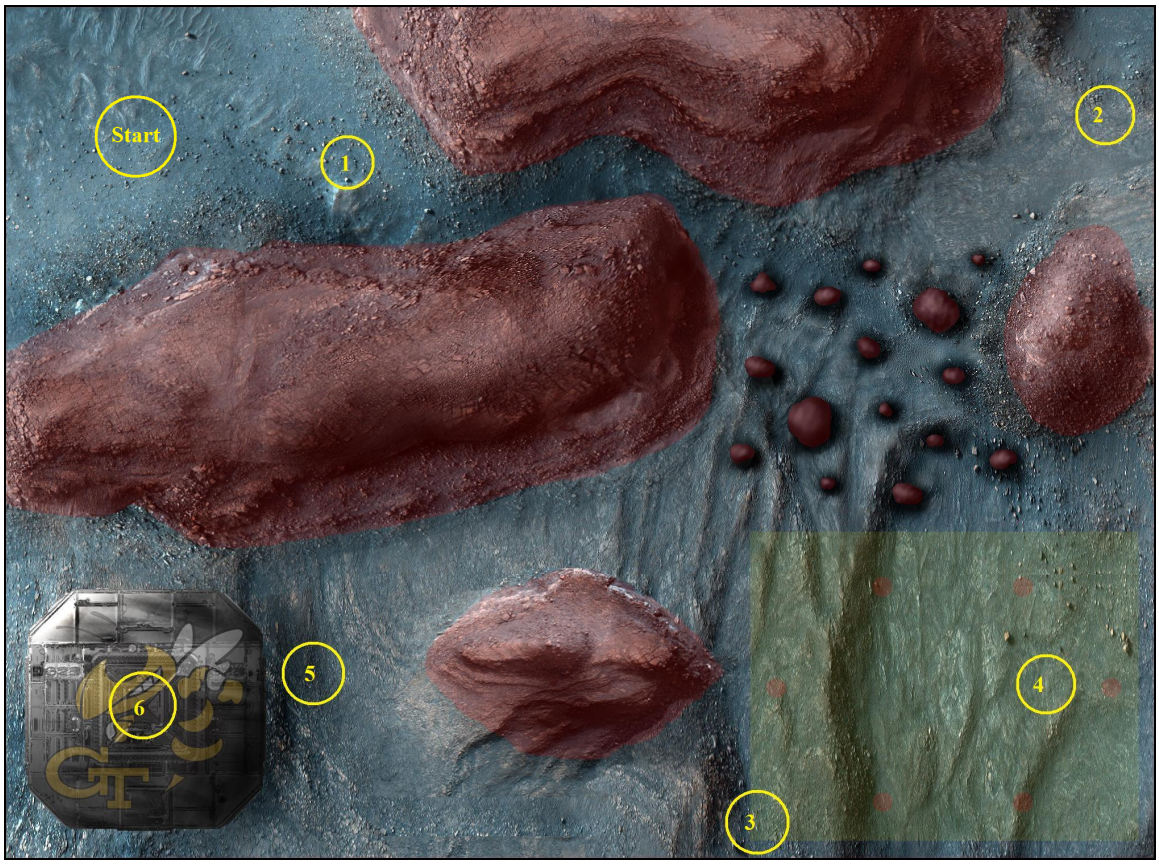
The year is 2030 and NASA has identified an asteroid that is on a collision course with Earth! In order to deflect the asteroid, the scientists require samples from its surface to determine its physical composition. They have sent a multi-robot team to collect samples from the asteroid's surface and bring them back to Earth for analysis. The robots managed to land on the asteroid successfully and were able to gather the samples. However, an unexpected pulse of electromagnetic radiation temporarily disabled the electronics on-board the robots, stranding them on the asteroid.

Based on your experience in networked control and multi-agent systems, you have been selected to lead a rescue mission. Using the beacons placed on the surface of the asteroid from the first expedition for navigation, your mission is to design decentralized controllers for the multi-robot rescue team so as to:

1. Navigate a team of 6 robots through the rough terrain of the asteroid
2. Locate and re-activate the 6 disabled robots from the first expedition
3. Bring both robot teams back to the platform (leave no robot behind) and get into a specific formation to wait to be picked up by an orbiting spacecraft

Section 2: Map of Terrain

To navigate the asteroid, robots must traverse between pre-defined waypoints, while avoiding collisions with obstacles and each other. The layout of the asteroid is given in the following figure, where the yellow circle on the top left labeled "Start" indicates where the robots begin, the numbered yellow circles correspond to waypoints which must be visited in order, and the parts of the terrain colored red correspond to obstacles which must be avoided. In addition, agents are not allowed to go outside of the map so the borders of the map are also considered "obstacles" which must be avoided.



Section 3: Simulation Environment

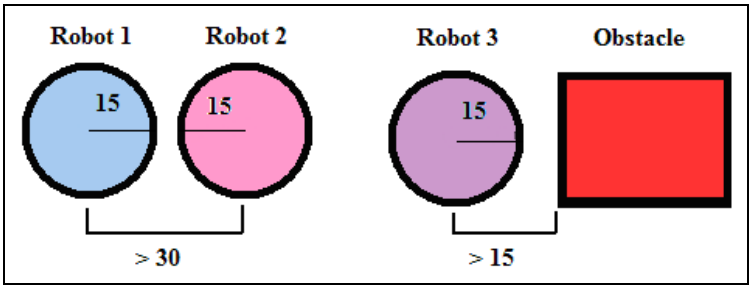
You will be running the decentralized control laws that you've designed for the multi-robot team in a MATLAB simulation environment, which will simulate the dynamics and sensors of the robots, as well as any interaction with the environment.

3.1: Collisions

While navigating through the environment, the robots must avoid colliding with each other and the environment at all costs. **A collision will destroy a robot and hence cause the mission to fail.**

- *3.1.1: Robot Frame*
  The robots used for this mission each have the shape of a circle with radius given by `agentRadius = 15`.

- *3.1.2: Agent-to-Agent Collisions*
  To prevent two robots from colliding with each other, their centers must be more than `2*agentRadius` away from each other at all times.



NOTE: To temporarily turn off agent-to-agent collisions, set the variable `AGENT_COLLISION_ON` to 0 in *simulator.m

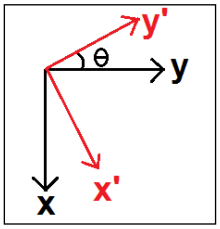
- *3.1.3: Agent-to-Obstacle Collisions*
  To avoid colliding with obstacles in the environment, the center of the robot must be more than `agentRadius` away from the edge of an obstacle at all times.

  NOTE: To temporarily turn off agent-to-obstacle collisions, set the variable `OBSTACLE_COLLISION_ON` to 0 in *simulator.m

3.2: Robot Sensing and Actuation

- *3.2.1: Types of Robots*
  The robots in your team can be classified into two types: a single leader robot, and the rest of which are follower robots. Both types of robots can sense other robots, obstacles in the environment, and have the same dynamics. However, the leader robot has access to information about the waypoints and can use it to navigate the team across the terrain.

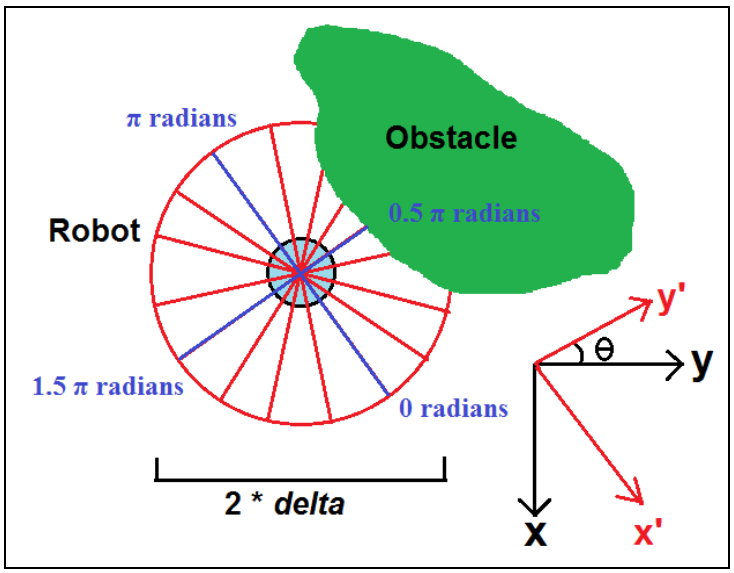
- *3.2.2: Relative Coordinate Frames*
  All sensing and actuation will be done within a relative coordinate frame that is unique for each robot. A robot's relative coordinate frame is simply the global coordinate frame rotated by a fixed offset angle θ that is unique to the robot.



- *3.2.3: Sensing Waypoints*
  Beacons placed throughout the terrain from the previous expedition act as waypoints to help you navigate. Only the leader robot is equipped to sense these waypoints and it is only able to determine its relative displacement vector from its center to the center of the waypoint.

- *3.2.4: Sensing Other Robots*
  Each robot is equipped with omnidirectional sensors for measuring the relative displacement vectors between its own center and the centers of other robots that are within `delta = 200` units. Thus, information flow amongst agents is given by an undirected delta-disk proximity graph.

- *3.2.5: Sensing Obstacles*
  The robots each have 16 sensors mounted pointing in different directions that detect the distance from the robot's center to the edges of obstacles that are within the sensing range of `delta`. A robot's obstacle sensor will return an array of 16 range readings, where the ith entry (i ranges from 0 to 15) will be the reading of the sensor pointing at the angle i/16*2*π within the robot's relative coordinate frame. Sensor angles that do not detect any obstacles will return `inf` (∞).



For the above image, the robot's obstacle sensor reading will look like:
`delta * [inf, inf, inf, 0.7, 0.54, 0.5, 0.8, inf, inf, inf, inf, inf, inf, inf, inf, inf]'.`

Furthermore, the range readings have a limited resolution of `0.02*delta`, meaning that if an obstacle is detected, the corresponding range reading will be rounded to the nearest multiple of `0.02*delta`.

- *3.2.6: Robot Dynamics*

  The robots can move freely on the plane and have single integrator dynamics. If the norm of the control vector exceeds the saturation limit of `uMax = 1000` (the maximum speed), then the length of the control vector will be scaled to equal `uMax`. Therefore, if the position of the ith agent is given by x_i and its control vector is u_i, then the dynamic are:

$$\dot{x}_i = \begin{cases} u_i & if\ ||u_i|| \leq uMax \\ \frac{u_i}{||u_i||} uMax & otherwise \end{cases}$$

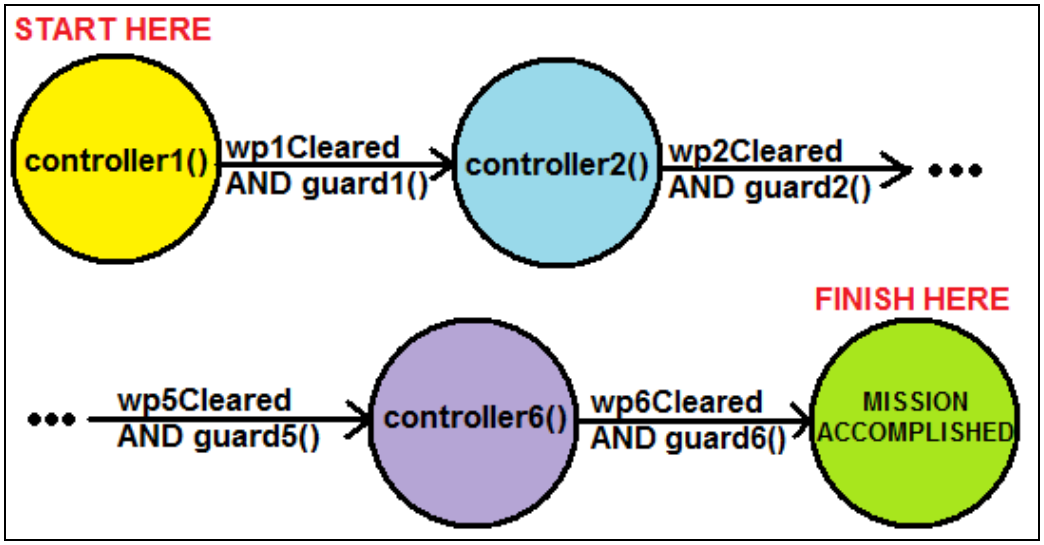
***NOTE: The dynamics of the robots are simulated using Euler integration with a fixed time step of `dt = 0.01`. Therefore, if a control law is not working as expected, try reducing the gain.**

NOTE: To temporarily turn off actuator saturation, set the variable `ACTUATOR_SATURATION_ON` to 0 in *simulator.m

3.3: Controller Implementation

- *3.3.1: Control Flow*

  There are a total of 6 waypoints to clear in the mission. You will be implementing a single decentralized controller, which all robots will execute, for each of the waypoints. You can also implement additional guard conditions for switching between the controllers, of which only the leader robot can evaluate. Therefore, the flow of control will be a hybrid system as illustrated in the figure below, where controller i will execute on all the agents until waypoint i has been cleared AND the additional guard condition that you have implemented for waypoint i returns TRUE for the leader. When both conditions are satisfied, all robots in the network will then switch simultaneously to executing controller i+1.



- *3.3.2: Source File*

  All the code that you implement will be located within a single m-file, named "lastname.m", where you should replace "lastname" with your actual last name. You can change the other simulation files as you wish for debugging purposes but note that in the end, all that you will be submitting is the "gtid.m" file and we will test it using the simulation files that were initially provided to you.

- *3.3.3: Controllers*

  Within the file "lastname.m", there are blank functions for you to provide your implementation of controllers 1 through 6. Each controller function looks like the following:

```
function [u,saveData] = controller1(uid,nbrData,wpData,
      obstacleData,missionData,saveData,delta,agentRadius,
      firstCall)

      u = [0;0]; % Give no control for now,
                 % this is for you to implement.
end
```

The parameters that you are given are summarized below (and are also described in the comments within the code):

- `uid`: The unique integer identifier of the agent that is executing this controller. The leader robot has identifier 1, while the other robots have identifiers 2, 3, and so on. This is used so that the same controller can act differently depending on which robot is executing it, such as having the leader robot go towards a waypoint, while all other robots do consensus.

- `nbrData`: A (M x 3) array, where M is the number of neighbors of the current robot. Each row corresponds to a relative displacement vector reading for one of the robot's neighbors. The ith row of `nbrData` is given by `[relativeX, relativeY, nbrUID]`, where `(relativeX,relativeY)` is the relative displacement vector pointing from the center of the current robot to the center of the neighbor robot. The entry `nbrUID` gives the unique identifier of the neighbor whose relative displacement you are measuring. Therefore, the robots are able to distinguish who it is that they are sensing in the network.

- `wpData`: This data is only available to the leader robot, and will be `[0;0]` for all other robots. For the leader robot, this is a (2 x 1) vector giving the relative displacement vector pointing from the center of the leader robot to the center of the current waypoint.

- `obstacleData`: A (16 x 1) vector of sensor readings of the relative ranges between the center of the robot and the edges of nearby obstacles. The details of how obstacles are sensed and what the sensor vector contains are described in Section 3.2.5: Sensing Obstacles.

- `missionData`: This is used to pass information to agents that are specific for clearing certain waypoints. The purpose of this parameter changes depending on the waypoint being cleared and will be described in Section 4: Clearing Waypoints.

- `saveData`: This parameter gives the values stored in memory for this particular robot in the form of a (10 x 1) vector. You can use this storage space on each robot however you please (such as keeping track of your past actions, waypoints visited, etc).

- `delta`: The maximum distance in which a robot can sense another robot or obstacle.

- `agentRadius`: The radius of the robot. A collision occurs between robots if their centers get within `2*agentRadius` of one another. A collision occurs between a robot and an obstacle if the robot's center gets within `agentRadius` of the obstacle's edge.

- `firstCall`: This value will be 1 if it is the first time that this particular robot has executed this controller. It can be used to know when to initialize the storage memory (`saveData`) of the robot.

Each of the controllers must be implemented such that they return two values to the simulator, both of which are summarized below:

- `u`: A (2 x 1) vector containing the calculated control signal for the robot. The dynamics of the robot are given in Section 3.2.6: Robot Dynamics.

- `saveData`: A (10 x 1) vector containing the updated contents of the robot's memory. The vector will be saved and passed to this robot again the next time the robot executes a controller. Note that the return value must be a (10 x 1) vector. Anything more will be truncated and anything less will be padded with 0's to form a (10 x 1) vector in memory.

- *3.3.4: Guard Conditions*

  Also within the "lastname.m" file, there are dummy functions for you to provide your implementations of guards 1 through 6. Each guard function looks like the following:

```
function [guardCleared] = guard1(nbrData,wpData,obstacleData,
      saveData,delta,agentRadius)

      % Set guardCleared to 1 to not impose any additional
      % conditions on the mode switches
      guardCleared = 1;
end
```

  The parameters passed to the leader robot for evaluating the guard condition are a subset of those passed into the controllers and so will not be described again.

  The guard function must be implemented to return a single value:

  - `guardCleared`: The value 1 if the user-defined guard (evaluated by the leader robot) is cleared, and 0 otherwise.

- *3.3.5: Implementation Constraints*

  The arguments to the controllers and guards have been set up in a way so as to force the robots to make their decisions using only local information and limited memory. **Do not try to cheat by using global or persistent variables, reading and writing to files, etc. Submitted m-files will be manually checked for this.**

Section 4: Clearing Waypoints

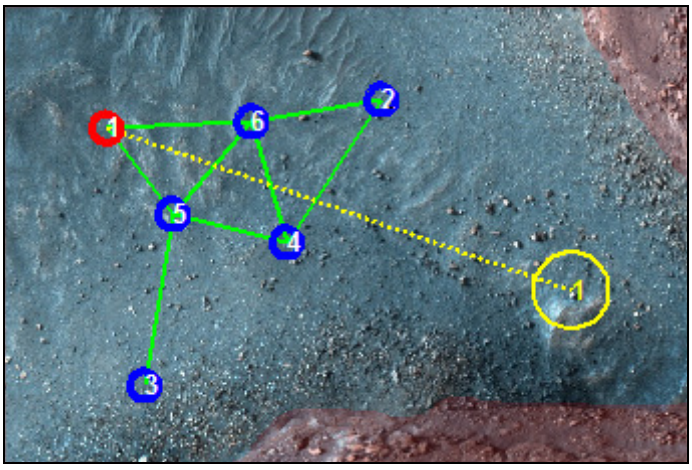
Navigating to and clearing each of the 6 waypoints corresponds to having the multi-robot team perform a series of 6 different tasks:

1. Go to waypoint without agents colliding
2. Squeeze through tunnel without colliding with terrain
3. Avoid multiple obstacles
4. Search an area for other agents
5. Split and merge around an obstacle
6. Drive agents into a pre-defined formation

In order to clear a waypoint, the center of a robot must be within `wpRadius = 40` of the waypoint's center. Furthermore, the information flow network formed by the agents must be a connected graph. Specific waypoints may require additional conditions to be cleared and are outlined below.

4.1: Waypoint 1: Go to waypoint

The robots start in the top-left corner of the map and are initially in the following configuration:



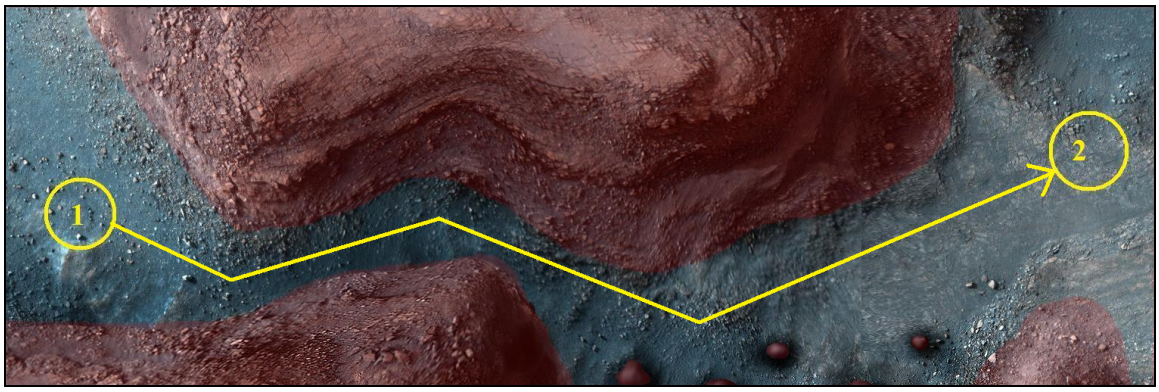
The purpose of this waypoint is for you to figure out how to drive agents to a waypoint, maintain network connectivity, and avoid agent-to-agent collisions.

No additional conditions are required to clear this waypoint.

No special mission data will be provided for clearing this waypoint. Thus, for all robots, `missionData = []`, the empty matrix.

4.2: Waypoint 2: Squeeze through a tunnel

The robots must navigate from one waypoint to another, while squeezing through a tunnel without colliding with the environment.



The purpose of this waypoint is for you to experiment with balancing the two goals of: avoiding collisions with the environment, and reaching a waypoint. Furthermore, you will get to try out different network topologies since it is near-impossible to navigate the robots through the tunnel while staying in a complete graph.

No additional conditions are required to clear this waypoint.

No special mission data will be provided for clearing this waypoint. Thus, for all robots, `missionData = []`, the empty matrix.

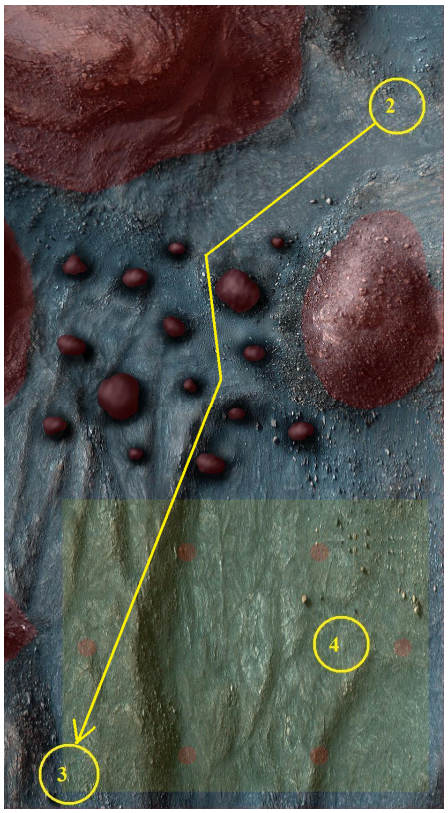
4.3: Waypoint 3: Avoid multiple obstacles

The robots must navigate through a field of scattered obstacles, while not colliding with any of them.

The purpose of this waypoint is for you to again find a balance between avoiding collisions with the environment and reaching a waypoint. You are allowed to take any path through the field so you may want to experiment to see which route is the easiest to navigate through.

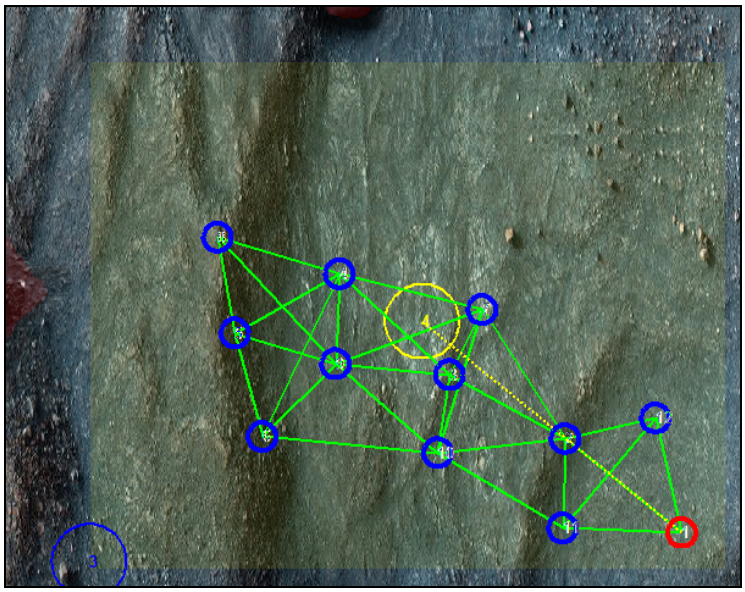
No additional conditions are required to clear this waypoint.

No special mission data will be provided for clearing this waypoint. Thus, for all robots, `missionData = []`, the empty matrix.



4.4: Waypoint 4: Search an area

The 6 robots in your rescue party must search the area highlighted in yellow for the 6 disabled robots from the previous expedition. When an active robot's center gets within `delta/3` of a disabled robot's center, it will be re-activated and join your network. Beacons are placed on the 4 corners of the yellow highlighted area, providing each robot with relative displacement readings to the 4 corners of the area that is to be searched.



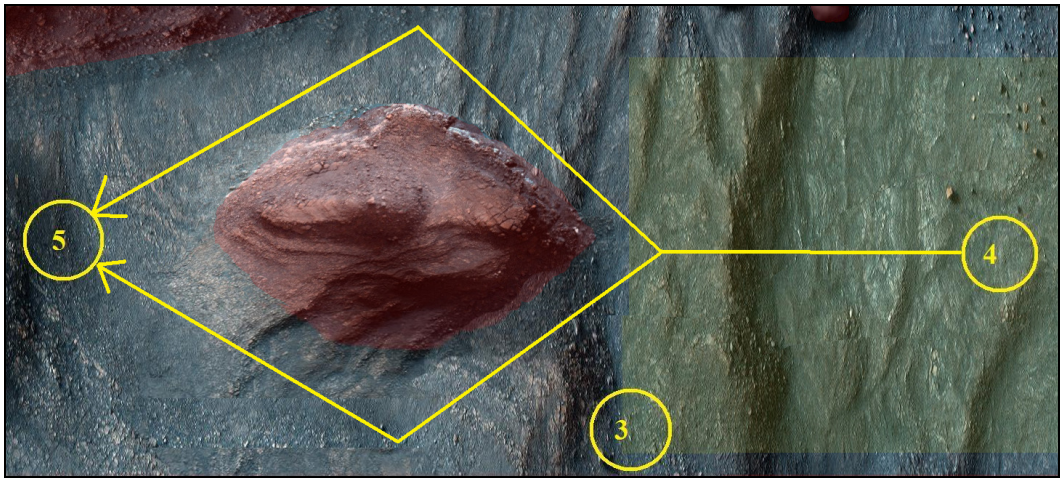
The purpose of this waypoint is for you to experiment with methods to search an enclosed space, and to handle agents dynamically joining the network.

The additional requirements for clearing this waypoint are that all 6 disabled robots must be re-activated, and the final network of 12 robots must be connected.

Beacons are placed at the four corners of the yellow rectangle that highlights the area to be searched. Each of the robots can sense their relative displacement vectors to each of the beacons. That information is contained in the parameter `missionData`, which will be a (4 x 2) matrix passed into each robot's controller, where each row gives the relative displacement vector pointing from the center of the robot to one of the corners of the yellow rectangle.

4.5: Waypoint 5: Split and merge around an obstacle

The 12 robots must navigate around a large obstacle by "splitting and merging", i.e., some of the robots must go to the left and some must go to the right of the obstacle.



The purpose of this waypoint is for you utilize the memory features of the robots so that robots which have "split" from the rest of the group will remember what direction they were initially heading in, and can "merge" back with the group after going around the obstacle.

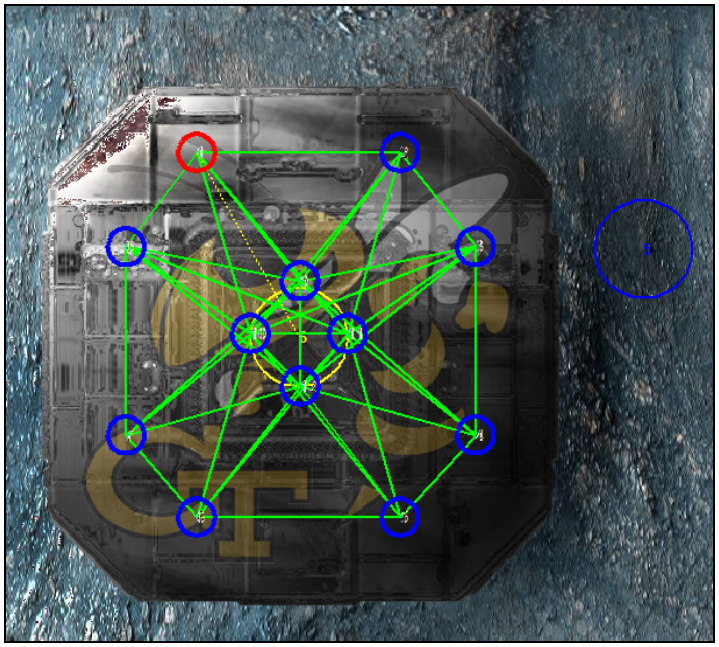
The additional requirement for this waypoint is that at least 1 robot must traverse from Waypoint 4 to Waypoint 5 by taking the route to the left of the obstacle, and at least 1 robot must do so by taking the route to the right of the obstacle. You are not allowed to cheat by having a robot take one route to Waypoint 5, go back to Waypoint 4, and take the other route to Waypoint 5.

***NOTE: This waypoint will clear within the simulation irregardless of whether you actually perform the "splitting and merging" maneuver. However, we will run each of your controllers and verify this manually.**

No special mission data will be provided for clearing this waypoint. Thus, for all robots, `missionData = []`, the empty matrix.

4.6: Waypoint 6: Get into a formation

The robots must all get into a pre-defined formation aboard the platform and wait for rescue. Beacons are implanted into the platform to tell each of the robots the relative displacements between themselves and each of the target locations where a robot is supposed to be. However, the robots are free to decide which robot goes to which target location, as long as there is one robot at each of the target locations.



The purpose of this waypoint is for you to execute formation control while avoiding agent-to-agent collisions, and to use assignment techniques to decide which robot to assign to which target location.

The additional requirement for clearing this waypoint is that each target location in the formation must have a robot touching it, i.e., each target location must have an agent whose relative displacement vector to it has distance less than or equal to `agentRadius`.

The target locations are marked by beacons on the platform and relative displacement vectors to them can be sensed by every robot. The matrix `missionData`, which is passed as an argument into every robot's controller, will be a (12 x 2) matrix where each row gives the relative displacement vector pointing from the center of the robot to one of the target locations on the platform.

Section 5: Getting Started

This section will provide you with a quick tutorial to get you started with the project.

1. Look for the file *lastname.m* and replace "lastname" with your actual last name.

2. Open the *lastname.m* file and write your full name in the header comments.

3. To get comfortable with writing controllers for the robots, let's start by writing a very simple controller that will drive the robots to the first waypoint, while ignoring any agent-to-agent and agent-to-obstacle collisions. Start by opening the *simulator.m* file and setting the values of both `AGENT_COLLISION_ON` and `OBSTACLE_COLLISION_ON` to 0. Be sure to save the file before closing it.

4. Open up *lastname.m* and insert the following code into `controller1()`:

```
function [u,saveData] = controller1(uid,nbrData,wpData,
obstacleData,missionData,saveData,delta,agentRadius,firstCall)

      if(uid == 1) % Action to take if robot is the leader
            u = wpData; % Move towards the waypoint
      else % Action to take if robot is the follower
            M = size(nbrData(:,1)); % M is # of neighbors

            % Let the follower robot's control be the sum of
            % relative displacement vectors between itself and
            % all neighbors, i.e., consensus equation
            u = 0;
            for i=1:M
                  u = u + nbrData(i,1:2)';
            end
      end
end
```

5. Save the file *lastname.m* and go into the MATLAB terminal. Set the current directory to the folder where all the simulation files are located and run the simulation by typing into the terminal: "simulator(@lastname)", where "lastname" is your last name.

6. A figure window should open showing the robots' initial positions. Press any key to start the simulation. The robots should execute the controller, where the leader robot will move towards the first waypoint while all other robots follow it. Once the robots reach the waypoint, they will stop moving since we have not implemented `controller2()`. Stop the simulation by pressing ctrl+C.

7. Congratulations! You just wrote your first controller! However, as you have noticed, the follower robots all end up on top of each other. Now, go back and modify the controller so that the robots will avoid colliding with each other and try again. Once you are comfortable with your solution, be sure to go back and turn on both agent-to-agent and agent-to-obstacle collisions in *simulator.m*.

Section 6: Saving and Loading the Simulation State

When trying different controllers for the robots, the simulation will stop upon an agent-to-agent or agent-to-obstacle collision. When debugging a controller, it would save time to be able to retry the simulation again starting from the previously successfully completed waypoint, as opposed to from the very beginning of the simulation. Therefore, the feature of saving and loading the simulation state was integrated into the simulator. Here's how it works:

1. Call simulator(@lastname) as usual. Suppose you clear WP1 but fail at clearing WP2...

2. A prompt will come up: "Would you like to save the simulation state to re-start from the previous waypoint [Y/N]? " Enter "Y" to save, or "N" to not save.

3. If you chose to save, another prompt will come up asking you for the filename to save the data under: "Enter filename to save as (without extension): " For example, if you type "myAttempt" (without the quotes), the data will be saved under "myAttempt.mat" in the current folder.

4. The simulation state (agent positions, agent memory contents, etc.) at the start of the last successfully completed waypoint will now be saved.

5. After modifying your controller, you can restart the simulation from the saved state by calling " simulator(@lastname, 'myAttempt') ". The optional 2nd argument to simulator.m is the string of the filename that you saved your simulation state under. Be sure to use single quotes.

*** NOTE: Only the agent positions, contents of the agent's memory, etc. are saved. Other simulation parameters, such as the flags for whether or not to ignore agent and obstacle collisions, are not saved. That way, you can toggle them on and off as needed while you work on your controllers for each of the waypoints.**

Section 7: Conclusion

This project will help reinforce the networked control concepts that you have learned in the course.

6.1: What to Turn In

You will submit via email (to Greg Droge at gregdroge@gatech.edu) a single .m file named *lastname.m*, where "lastname" corresponds to your last name. The file should be such that one can run your implemented controllers by calling simulator(@lastname) within MATLAB.

6.2: Hints

1. The first two waypoints are the hardest to clear since you are just starting, so don't get discouraged if it takes you a while to figure them out.

2. If you are having a hard time getting started, try first turning off agent-to-agent and agent-to-obstacle collisions. Once you get used to the interface and are able to move the agents to a waypoint, then start worrying about collisions. It is acceptable to turn in code that only works with collisions turned off, for partial credit.

3. The choice of network topology is very important.

4. There is no time limit for the robots to complete the mission. If you find your robots over-reacting and crashing into obstacles or other robots, try decreasing the gain in your controller. Try increasing the gain if the action is not large enough.

5. Although you are allowed to use any method to clear each of the waypoints, the ideas that we've learned throughout the class are very powerful and can be used to complete each of the waypoints with only minor modifications.

6. The problems addressed in this project are all very common multi-agent tasks and there has been a lot of published literature on these subjects. If you are stuck, try looking into other people's works for some inspiration.

6.3: Questions

The amount of programming in this project is *actually* not that much. If you feel uncomfortable with MATLAB, there are a lot of good resources/tutorials on-line to learn the basics.

For questions regarding the project, email Greg Droge at gregdroge@gatech.edu

Good luck, the future of humanity rests in your hands!