

Distributed Intelligence in Autonomous Robotics

Assignment #1

Out: Thursday, January 16, 2003

Due: Tuesday, January 28, 2003

The purpose of this assignment is to build familiarity with the Nomad200 robotic simulation environment we will be using this term for your homeworks in this class. This exercise involves downloading the simulation software and example code, and experimenting with various aspects of the system. You should learn how to run the simulator, how to run robot code, how to create your own test environments, how to recompile robot code, and how to obtain screen shots of the simulator.

This simulation system operates under Solaris Unix (any of the Sparcs in the CS lab should work) using C or C++ programs. For parts of this assignment, you will need to obtain a screen dump. Instructions are provided for how to obtain and print screen dumps. For this assignment, you should turn in the output requested below in the section entitled “TURN IN THE FOLLOWING” on page 4.

**** YOU MAY HELP EACH OTHER AS NEEDED TO COMPLETE THIS ASSIGNMENT. HOWEVER, EVERYONE SHOULD TURN IN THEIR OWN SCREEN DUMPS OBTAINED FROM RUNNING THE EXAMPLE CODE IN THEIR OWN DIRECTORIES. DO NOT TURN IN COPIES OF SCREEN DUMPS THAT YOU DID NOT GENERATE YOURSELF IN YOUR OWN UNIX AREA.**

1. Download the Nomad200 simulator and example robot control code.

- Using your favorite web browser, go to the course handouts web page at <http://www.cs.utk.edu/~parker/Courses/CS594-spring03/handouts.html> and download initial code package. Save the software (which will be in a file called “InitialCode.tar”) in your home Solaris Unix directory.
- Unpack the simulator and example code software:

➤ `tar xvf InitialCode.tar`

This command creates a directory in your area called “Nomad200” with two subdirectories – “client” and “server”. In the client subdirectory is another subdirectory called Examples. This subdirectory has example robot control code.

2. Run the Nomad200 Simulator

- Create three xterm windows, which will be used later:
 - `xterm &`
 - `xterm &`
 - `xterm &`

- Connect to the server directory:

➤ `cd Nomad200/server`

- Run the simulator from one of the xterm windows:

- `./Nserver &`

You should see 4 windows open. The titles of the windows are: “Map”, “Robot:Nomad(1)”, “Options”, and “Options”. The “Map” window gives the global view of the environment, the “Robot:Nomad(1)” gives information specific to robot 1, the topmost “Options” window displays values of the infrared sensors for robot 1, and the lower “Options” window displays values of the sonar, laser, and proximity sensors for robot 1.

- With the simulator running, you can now experiment with the example robot control code in the following steps. If you want to quit the simulator and come back later, select “Quit” from the “File” pulldown menu in the “Map” window.

2. Run example robot control programs

Attached to this handout is a description of the example robot control code. Read these descriptions for more detail on what these example programs do.

For all of the example code, there are README files that indicate how to compile and run the code. The following walks you through the Wander example. A similar approach will work for the Multiple_behaviors example (be sure to read the README file for specific instructions).

2a. Wander robot code example

- Connect to the Nomad200/client/Examples/Wander subdirectory.
- Be sure you opened 3 xterm windows from step 1 above, and that Nserver is running.
- Open an example environmental map by selecting “Open Map” from the “File” pulldown menu in the “Map” window. Select the file “map1.txt” to open (from the Nomad200/server/Sample_maps subdirectory). You should see a map with walls, obstacles, etc., open in the “Map” window and in the “Robot:Nomad(1)” window. Zoom out by selecting “Zoom out” from the “View” pulldown menu in both the “Map” and “Robot:Nomad(1)” windows. (After you select zoom out, you will need to click in the window to execute the zoom.) Practice changing the view in both windows, zooming in and out, sliding the display, and so forth.
- In one of your xterm windows, connect to the Wander code example in the client directory:
 - `cd Nomad200/client/Examples/Wander`
- Since we want to trace the path of the robot, select one of the “Robot trace” options (Solid or outline) under the “Show” pulldown menu in the “Robot:Nomad(1)” window. When the robot begins moving in the next steps, you should see a trace of the robot’s path in the “Robot:Nomad(1)” window.
- Run the sample code that uses sonar to have the robot wander through the environment:
 - `./sobounce 1`

(In the above command, ‘1’ stands for robot 1. When there are multiple robots running, you replace “1” with the number of the specific robot.)
- Observe the robot moving through the environment and note its behavior over several minutes.

- To halt the robot, enter `<ctrl>-c` in the window where “sobounce” is running. To refresh the display, select “All” under the “Refresh” pulldown menu in the “Robot:Nomad(1)” window.

2b. Multiple behaviors robot code example

- Perform a similar series of steps to the above to experiment with running the Multiple behaviors robot code example (in `Nomad200/client/Examples/Multiple_behaviors`). Be sure to read the README file in that directory for specific instructions. For this example, use the sample map “map4.txt”.
- In this case, there are 3 examples of robot control code that all accomplish the same task (ex1.c, ex2.c, ex3.c). Run each of these examples and examine the code to get a feel for how the robot control is generated.

2c. Multiple robot formations code example

- Read the attached document on how to run multiple robots in the Nomad 200 simulator.
- Read and follow the instructions in the README file in `Nomad200/client/Examples/Multiple_robots`.
- Run the robot formation code example with 4 robots.

3. Create a customized map

- Clear out the simulator environment by selecting “New Map” under the “File” pulldown menu in the “Map” window. The map of walls and obstacles, etc., should disappear. You can now create your own map by using the “Obstacles” pulldown menu. Experiment with adding obstacles and polygons to create your own map. You can save maps by selecting “Save Map” or “Save Map As” under the “File” pulldown menu.

HOW TO OBTAIN A SCREEN DUMP:

Obtaining screen dumps. To obtain screen dumps, enter the following command in an open xterm window:

- `xwd -root -out screen.xwd`

This command creates a screen dump and puts it into the file called “screen.xwd”. You can crop, resize, and print by using the “xv” utility:

- `xv screen.xwd`

Right-click the mouse to get the xv controls. Experiment with changing the image size and cropping to ensure that the simulator portion of the screen is included and legible in the printout. To print the image, select the “print” button and (perhaps) the landscape mode, then select “grayscale”, then OK. Enter Quit to exit. More information on xwd and xv can be found using the unix man pages (i.e., “man xwd” or “man xv”).

TURN IN THE FOLLOWING:

I. FROM PART 2a: A screen dump of your robot running the Wander example in map1.txt, showing the robot trace of its path over several minutes. Make sure that the complete map is visible in the “Map” and “Robot:Nomad(1)” windows. Print out and turn in a hardcopy of this screendump.

II. FROM PART 2b: A screen dump of your robot running the Multiple_behaviors example code in map4.txt, showing the robot trace of its path over several minutes. Print out and turn in a hardcopy of this screendump.

III. FROM PART 2c: A screen dump of your robot running the Multiple_robots example code in an empty map (i.e., no obstacles), showing at least one robot trace of its path for path following. Print out and turn in a hardcopy of this screendump.

IV. FROM PART 3: To become familiar with generating your own test environments, create your own customized map of moderate complexity (completely enclosed, at least 8 rooms and 4 obstacles). Turn in a screen dump of the robot running either the Wander or the Multiple_behaviors example in your new environment.

Instructions on Using Multiple Robots in the Nomad 200 Simulator

These instructions will get you started on how to simulate multiple robots in the Nomad200 Simulator.

Creating setup files for multiple robots

First, you need to create setup files for each robot. My recommendation is to name the robots with unique names, to minimize confusion. To do this, you need a .setup file for each robot. For example, if you have 4 robots named edith, grace, alex, and ada, then you would have four files called edith.setup, grace.setup, alex.setup, and ada.setup. These setup files would be identical to the existing robot.setup file, except for changing the following parameter:

- `wind_name`: This is the name of the simulation window that will appear for this robot. In the robot.setup file you have been using, the `wind_name` value is “Nomad”. You should change this to one of your robot names (e.g., Edith, Grace, Alex, or Ada).

You can make each robot unique in some way by changing other parameters in the setup file, as you wish.

Creating multiple simulation windows

Next, you need to create simulation windows for multiple robots. There are two ways to do this. First, after you run Nserver, you can “create robot”, using the Control pulldown menu in the main window to add additional robots. This will then prompt you to select a robot setup file for the next robot. You would then select the setup file you have created above for the next robot you want to create. You can “create robot” multiple times to keep adding robots.

Alternatively, you can initiate the world.setup file with the names of the robot setup files you want to have initiated upon startup. You do this by setting the following parameter in world.setup:

- `setup_files`: This parameter provides a list of the setup files to start upon invocation of the simulator. You can add a list of robot setup files, separated by whitespace, to automatically start up multiple robots when you invoke the Nomad 200 simulator. For example:

```
setup_files = edith.setup grace.setup alex.setup ada.setup
```

will automatically start up 4 robots when you invoke the Nomad 200 simulator.

Unique robot numbers

The Nserver automatically numbers the robots sequentially as they are created, so when you create a second robot, its number “(2)” will appear in the window where the robot “(1)” appears now for your first robot. This is the number that must be used in the `connect_robot` function call (described below) to connect robot control code to that specific robot.

Running control code for multiple robots

In this simulator, you have one executable robot control code for each robot. To run code to control each robot, it is easiest to open a new xterm window for each robot. Then, in each window, you run the control code for one robot; the code can be different for different robots (e.g., “leader” code for one robot and “follower” code for another robot). The control code will be almost identical to the control code for single robots, with two differences, as follows.

First, be sure there is no call in the robot control code to the function “`server_is_running()`”. For some reason that I do not understand, this function call interferes with running multiple robots.

Secondly, the code for each robot will need to know which robot to connect to, based upon the unique robot number assigned by the server. To do this, you will need to pass in a parameter to your code that will be used by the “connect_robot” function call, so that the code will know which robot to connect to. For example, you can run sobounce for two robots by entering “./sobounce 1 &” and then “./sobounce 2”. The first command will connect to robot with the ID of 1, and the second will connect to the robot with the ID of 2.

Getting rid of a robot

To get rid of a robot, you can select the “destroy robot” command from the Robot pulldown menu of the specific robot you want to eliminate. The other robots will remain functional.

Inter-robot communication

In many multi-robot applications, you want robots to be able to communicate with each other by sending messages. To simplify this process, you will be provided with code that enables multiple robots to communicate in the Nomad 200 Simulator. The full details of how this inter-robot communication process works will be explained in class. The following documentation gives the basics.

The example code subdirectory Multiple_robots contains the following files (among others):

- communication_server.c // source code for communication_server
- communication_server // executable version of communication_server
- inter_robot_communication.h // include file for robot control code
- socket_includes.h // include file for robot control code
- testcomms.c // example code that uses communications

To use this code, you need to run the communication_server in the background with a specified port number (of your choice), as follows:

```
communication_server -port xxxx &
```

where xxxx is the port number you have selected, such as 7654.

Your code will access the communications server by entering this same port number. For example:

```
test_comms -portc xxxx n
```

where xxxx is the same port number specified when you invoked the communication_server, and ‘n’ is this robot’s unique integer ID. [Note that the parameters are ‘port’ for the communication_server and ‘portc’ for the robot control code.]

Your robot control code initializes communications by calling the “init_communication()” function. It can then receive messages from other robots by using the “listen_to_robot()” function, and will send messages to other robots by using the “talk_to_robot()” function. To access these functions, your robot control code should include inter_robot_communication.h and socket_includes.h, along with other system header files as shown in testcomms.c. The file testcomms.c gives examples of using these functions.

There are two types of messages robots can send: point-to-point and broadcast. In point-to-point, the communication_server relays messages only to the specified robot. In broadcast, the communication_server relays messages to all other robots. To send a point-to-point message, use the following function call:

```
talk_to_robot(‘P’,’ID’,msg);
```

where 'P' refers to point-to-point, 'ID' is a character representation of the robot's unique number, and "msg" is the message that you want to send.

To send a broadcast message, use the following function call:

```
talk_to_robot('B','0',msg);
```

where 'B' refers to broadcast, '0' is a dummy place holder, and "msg" is the message that you want to send to the robot. The communication routines expect "msg" to be a character string ended by the usual end-of-string character '\0'.

The format of "msg" is completely up to you. If the message contains multiple fields, then you will need to put in delimiters so that the receiving robot's code can decipher the message. If you have multiple types of messages, then you will probably want to include a character that specifies the message type at the beginning of the message. Also keep in mind that it is often helpful to know which robot sent which message. In this case, it is handy to include a send-robot-ID field within your message so that the receiving robot can know who sent the message. An example message for broadcasting x,y coordinates from robot '3' to all other robots would be a character string as follows:

```
"C31025$984"
```

where 'C' is the message type ID that we have given this message (meaning "my current coordinates coming" in this example), the '3' is the ID of the sending robot, '1025' is the current x position of robot 3 (converted to a character string), '\$' is the delimiter between the x and y values, and '984' is the current y position of robot 3 (converted to a character string). There is an implicit end-of-string character '\0' at the end of this message. Again, refer to the testcomms.c example code for how to compose such a message.

When you want your robot control code to receive a message from another robot, you use the following code:

```
msg=listen_to_robot()
```

which returns the messages currently in the buffer for this robot. Note that this message may be the concatenation of multiple messages that have been sent to this robot. Your code will have to parse the message to obtain the information held therein. This message parsing is essentially the reverse of what the sending robot control code did to generate the message in the first place.

Also keep in mind that if another robot is sending the same message repeatedly (or with updated versions of the message, such as relaying its own current position as it moves), then the first message string in the buffer will be the first message that the other robot sent. If you are only interested in the most recent message from a robot, then you'll need to process the message string until you reach the end of the message. If more than two robots are sending messages to each other, all messages will go into the same message buffer.

Refer to the testcomms.c or follower.c example code for how to process the messages.

Description of Example Robot Control Code

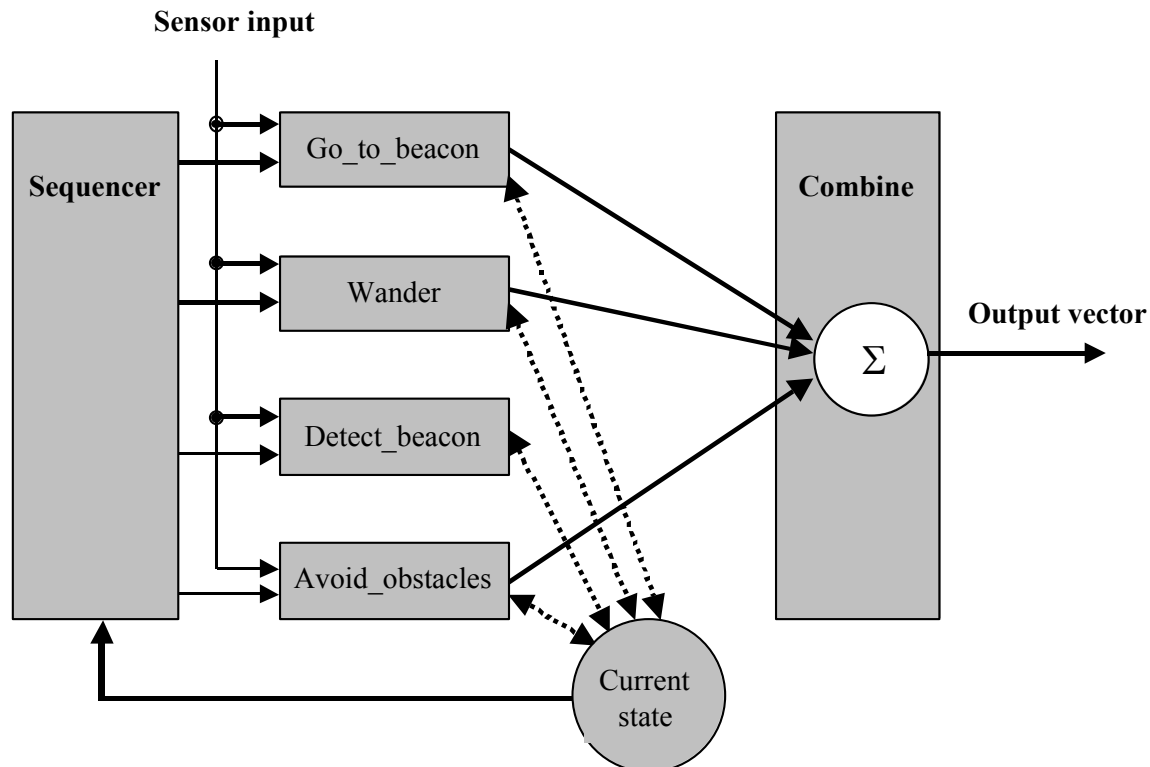
A. Wander

This code uses sonar sensors to enable the robot to wander around and avoid obstacles.

B. Multiple behaviors

Robot control code for behavior-based autonomous robots is usually built up incrementally. This example code enables a robot to execute multiple behaviors simultaneously. These behaviors are: `avoid_obstacles`, `wander`, `detect_beacon`, and `go_to_beacon`. The robot control is based upon motor schemas, with outputs of individual behaviors being vectors that are cooperatively summed to generate the resultant output control vector for the robot. The behaviors that are active at each point in time are determined by a sequencer that invokes the proper behaviors during the task. For example, the `go_to_beacon` behavior is only activated after the `detect_beacon` has actually found a beacon.

This robot behavior causes the robot to wander around avoiding obstacles (similar to `sobounce.c`, except that the “wander” behavior is separated from the “avoid obstacles” behavior) until it detects a beacon (which is like a “goal”). The robot then goes to the location of the beacon using the `go-to-beacon` behavior. Once the beacon is reached, the Sequencer causes the robot control code to exit. The following diagram gives the overall architecture of this robot control code.



Simulation of beacon. Often, robot simulators do not provide built-in capabilities to test all functionalities of a physical robot. In these cases, we have to build in new functions to allow us to simulate other robot capabilities. This can be done by changing the simulator itself (which can be rather involved), by mimicking additional capabilities using existing simulator functions, or through simple function calls. In this example, we simulate a robot sensor that can detect the position of a beacon whenever the robot is within a specified range of the beacon. This simulated beacon sensor assumes that obstacles in the environment are “short” (e.g., short walls), and that a beacon can be detected over the obstacles (e.g., a beacon on a pole). This simplifies the sensor simulation by eliminating the need to compute visibility from the robot to the beacon due to intervening obstacles.

The “initialize_beacon” function accepts the (x,y) coordinates of a beacon and, in order to aid the human user’s visualization of the beacons, draws a small circle in the simulated environment to visually mark the location of the beacon. The Nomad200 function “draw_arc(x,y,100,100,0,3600,1)” is used to mark the beacon position in the robot window.

The function called “detect_beacon(range)” provides the (x,y) position of the beacon if it is within a distance of “range” from the robot’s current position. If the beacon is not within range, then the function returns 0. For this example, the detection range is set to 2000 units.

Avoid obstacles. The perceptual schema for this behavior is the detection of a nearby obstacle in the direction of motion of the robot. The motor schema for this behavior is the generation of an output vector that turns the robot away from the obstacle. These capabilities are similar to those provided within the “sobounce” code for detecting and avoiding obstacles, except that instead of generating “vm” commands, the code generates an output vector giving the desired direction and magnitude of motion.

Wander. The perceptual schema for the wander behavior is a potential field that occasionally changes the robot motion by some random direction and velocity amount, within pre-specified turning and velocity ranges. The motor schema for this behavior is the generation of an output vector that turns the robot in the specified random way.

Vector combination. Output vectors from multiple behaviors are summed to generate an output vector by multiplying the individual behavior vectors by a given gain matrix, and then using vector addition.

Go_to_beacon. Here, the beacon position returned by the “detect_beacon” behavior is the robot’s perceptual schema, and the go_to_beacon motor schema generates an output vector that turns the robot toward the goal.

Behavior sequencing. The Sequencer module activates the behaviors as follows:

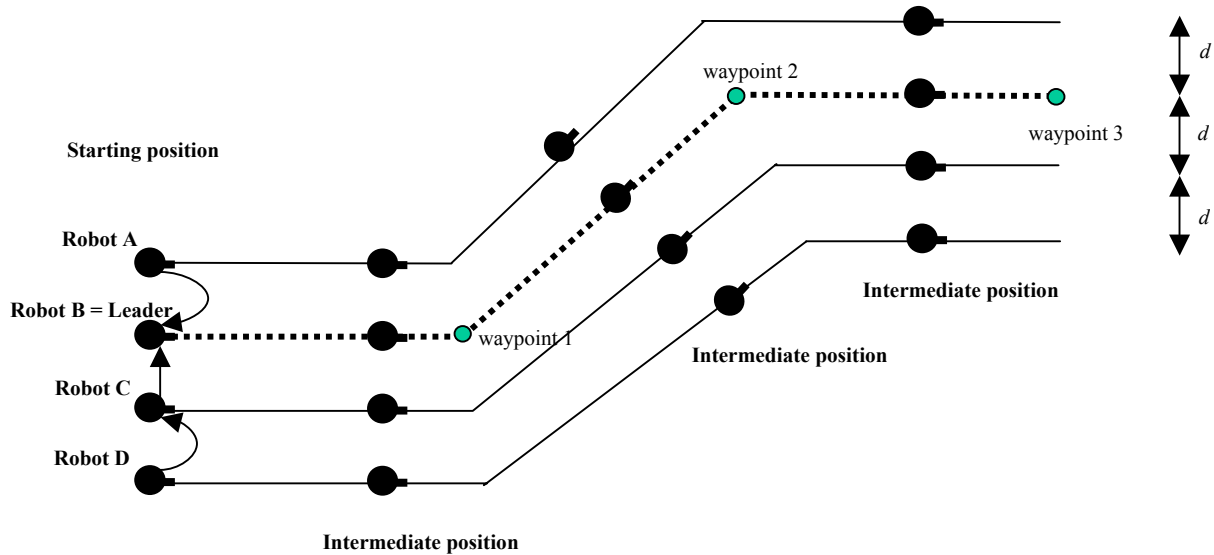
- Initially, “wander”, “avoid_obstacles” and “detect_beacon” are activated.
- When “detect_beacon” has found the beacon, “detect_beacon” and “wander” are deactivated, and “go_to_beacon” is activated.
- When the beacon position is reached, all behaviors are turned off and the code exits.

Beacon locations. Some beacon locations are easier for the robot to reach than others. Experiment with various beacon locations and their consequences for the robot control code. In particular, test out beacon locations with the following characteristics:

- I. Beacon within range of the starting robot position.
- II. Beacon out of range of starting robot position, but with no obstacles between the beacon and the robot when the beacon is first detected.
- III. Beacon out of range of starting robot position, and with “short, simple” obstacles between the beacon and the robot when the beacon is first detected.
- IV. Beacon out of range of starting robot position, and with “long wall” obstacles between the beacon and the robot when the beacon is first detected.

C. Multiple robots

In this example, the robot control code that allow 4 robots to move in side-by-side formation from one waypoint (i.e., x,y location) to another. The robots maintain a perpendicular distance d between them. Only one of the robots (the “Leader” robot) knows the waypoints to be followed. The rest of the robots must move in order to stay in formation with a specific reference neighbor. The following diagram illustrates this objective:



In this example, Robot B is the leader robot, and is the only robot that knows the location of the waypoints. The Robot B behavior for moving from waypoint to waypoint is the same as the go-to-beacon behavior from the `Multiple_behaviors` sample code.

Robots A, C, and D must make movements based upon the current location and heading of their own reference neighbor robot. In this example, the assignment of reference robots is as follows:

Robot	Reference Robot
A	B
C	B
D	C

In other words, Robots A and C move based upon the current position of Robot B, while Robot D moves based upon the current position and heading of Robot C. Rather than trying to sense this position and heading information using the sensors, we will use communication to simplify the approach. Thus, robots must communicate with each other to share their own current position and heading information with their teammates, so that this information can be used for velocity and steering control in robots A, C, and D. When the follower robots A, C, and D receive the current position and heading information from their reference robot, they calculate their own desired position in the formation as the location a distance d from the reference robot, perpendicular to the heading of that reference robot.

To simplify this control code, the following assumptions/constraints are made:

- There are no obstacles, so robots do not need behaviors to avoid obstacles.
- Robots do not move backwards. If a robot is in front of its desired position (as determined by examining the current position and heading of its reference robot), then it sits still (and perhaps rotates in place) until the desired position is in front of the robot.

- The code does not handle turns of more than 45° . Turns of much more than 45° would mean that neighboring robots are blocking the path of the leader at the point of the turn, which greatly complicates the formation control.
- The leader robot does not attempt to monitor the progress of the follower robots. That is, Robot B will just move through the series of waypoints provided to it and will stop when the final waypoint is reached without paying any attention to the other robots (other than to communicate its own position and heading information).
- The velocity control for Robots A, C, and D is a function of the distance between the robot's current position and its desired position. The further away the desired position is, the faster it moves. This helps the follower robots keep up with the leader.
- Robots are in the proper formation at the beginning of the task, facing in the direction of their next waypoint.

In this example, the following waypoints and parameter settings are used:

Waypoint/Parameter	Value
Starting position	$(x,y) = (0,0)$
Waypoint #1	$(x,y) = (2000,0)$
Waypoint #2	$(x,y) = (5000,1400)$
Waypoint #3	$(x,y) = (8000,1400)$
Inter-robot distance, d	500