Studying Social Interactions using Swarm Robotics*

Irme M. Groothuis

Department of Knowledge Engineering, Maastricht University

June 15, 2014

Abstract

In this paper, a multi-robot framework is proposed in which robots can be used to study the social behaviour. To explore the possibilities for designing the proposed framework, different robotic platforms and available simulators are studied. The best set-up is chosen and an appropriate interaction strategy is implemented which captures the main properties of a chicken game, as a simple game-theoretical model, for a multi-robot system. Various experiments are conducted which are all implemented in the *Stage* simulator with robotic platform *Turtlebot*. The experiment results confirm that the proposed framework can be helpful in validating and studying models of social behaviour.

1 Introduction

Studying social behaviour has a very long history, and in behavioural sciences various models are suggested to study social behaviour. Typically agent-based simulators or real human experiments are used to study/validate these models. However, using robots, can help with the calibration of these models, where the physical environment is supposed to effect the interactions between robots.

This research aims to implement an N-player Prisoner's Dilemma (PD). The N-player PD can be used to represent the behaviour of selfish and altruistic individuals in a society. Using robots and simulators one is able to model this more precisely and so one can build models of the evaluation of cooperation in society. This is needed as the physical environment is expected to influence behaviour.

This paper mainly focuses on the scenario that robots wander around randomly in an unknown environment and according to the predefined strategies, either cooperation or defection, interact with each other whenever they meet each other by chance. This experiment is further extended to large groups of robots and both the microscopic and macroscopic behaviours of the group are studied.

The advantage of using robots lays in the fact that the robots' strategy or mindset (either selfish or altruistic) can be easily changed, and so it is easier to study to effects of one individual's behaviour on the group as a whole.

This research seeks to answer to the following questions:

RQ 1 What is an appropriate two-robot experimental setting which can represent the simple PD problem?

RQ 2 How to extend the setting designed for the two-robot PD to a Swarm Robotic experiment?

RQ 3 What is the effect of structure of the

*This thesis was prepared in partial fulfilment of the requirements for the Degree of Bachelor of Science in Knowledge Engineering, University of Maastricht, supervisors: Prof. Gerhard Weiss, Prof. Karl Tuyls, Phd. Candidate Bijan Ranjbar-Sahraei

environment on swarm outcomes?

RQ 4 Can robots be used to study social behaviours?

2 Related Work

Many researches have focussed on behavioural robotics, for example in [4] it is stated that both robotic and agent based simulation are necessary to do good research as they are complementary. Doing robotic research allows one to better calibrate a model to the real world, as some problems will not come up in simulation, but might come up in real life. An example given in this paper is hardware failure.

Birk and Wiernik in [3] conclude that evolution and trust can emerge in larger groups of robots if the right strategies are used. They used a strategy called Justified Snobbism (JS). A robot using JS will cooperate slightly more than the average if a non-negative pay-off was achieved in the previous round. If a negative pay-off was received than it will cooperate exactly average. This strategy is at least as cooperative as the average, and allows cooperation to evolve. Experiments are run on a swarm of 20 robots.

Paper[2] studies the usage of AI experiments in different fields of studies, in this case economics. Using a small set-up with robots, lamps and feeding stations they study the behaviour of the robots as the parameters change. In general they conclude that the robots prioritise recharging energy above free time. This can be linked to the economic principle of behavioural resilience and elasticity of demands. If the cost of charging itself would increase with a certain percentage, charging will decrease by a smaller percentage, but if the same were to happen for free time, it would decrease far more.

In [5] it is stated that robots are useful when studying self-organisation, communication and the evolution of cooperative and competitive behaviour. It also states that studies using robots have generated new theories that can be tested on live animals. Lastly, it states that robots can be used in experiments where the physical environment or properties of the physical environment are likely to influence the outcome of social behaviour.

This paper, however uses robot simulation for social studies, and tries to validate the usage of robots in such research.

3 Preliminaries

In this section we introduce some of the preliminary topics which have formed the foundations of this work.

3.1 Prisoner's Dilemma (PD)

In the prisoner's dilemma, two players (A and B) have the option of either cooperating with the other or defecting. The players have to decide simultaneously what to do and they do not know their opponents decision. The resulting pay-off is shown below, where A is the row-player, and C and D represent the action of cooperating or defecting:

	C	D
C	5/5	0/20
D	0/20	1/1

The dilemma is when defecting has the highest pay-off if the opponent cooperates.

3.2 Chicken Game

Even though the chicken game and the iterated prisoner's dilemma are not the same, they both represent a conflict-based two player game in which defecting only results a higher pay-off if the other player cooperates. The chicken game was therefore used to represent a situation similar to the prisoner's dilemma. In this game, two players meet each other while travelling on a road. Each player can either decide to 'chicken' and drive slowly and make room for the opponent to pass, or they can just boldly go on their way, which may result in a collision if both decide to do so. Going slowly and providing space for your opponent costs more time and fuel, but it guarantees no collision. Doing nothing and just continuing driving ensures you will be quicker than the opponent, but brings the risk

of collision. Driving safely is seen as cooperating, while driving fast is seen as defecting.

3.3 Definitions

Here we provide the definition of game theoretical terms with respect to our experimental setting. We define a *Game* in which robots move randomly in the environment and interact when they meet each other. The definitions in this game are as following

- **Player**: A robot in the simulator
- **Strategy**: Either defecting or cooperating.
  - **Cooperation**: Moving out of the way of the opponent, so to chicken
  - **Defection**: Not moving to provide space for your opponent to pass, or to be brave.
- **Fitness**: The average speed of the robots: The fitness at interation k is given by:

$$F = \sum_{i=1}^{k} \Big(P(i+1) - P(i) \Big) \Delta t \quad (1)$$

  Where P(i) is the robot's position at iteration i, and Δt is the time step.

In this game, the microscopic behaviour refers to the individual's behaviour while the the macroscopic behaviour refers to the group's behaviour.

3.4 Robotic Platform

Two types of robots were considered as the platform for this research: TurtleBots and E-pucks

E-pucks

E-pucks are small robots that can be programmed using C. They are equipped with a camera, wheels, sensors and multiple LEDs and a microphone. The user directly programs with the input from the robots, manually managing all the input from the sensors etc. The advantage of e-pucks is that they are small and easy to manage and one is in direct control of the robot. However, code running on a simulator may not work on a real e-puck.

Turtlebots

Turtlebots are bigger robots that are equipped with various wheels, a laptop and an Xbox kinect sensor. They are programmed using ROS. This allows the user to run the same program on other robots using Robot Operating System (ROS). Code that will be run on the simulator should also work on the actual robots. The Turtlebots do require more high level programming, but the ROS interface allows for a more sophisticated research/program.[1]

Comparison

In this paper the Turtlebots were used, as the ROS-platform allows the user to use the same code on both simulators and robots. Furthermore, ROS allows the user to easily manage all input and write code to use as input or create a server to communicate with all robots.

3.5 Simulation Environment

For this thesis stage was used as the main simulator. Turtlesim and STDR were also briefly used in the research, but the main work was done using Stage. The main advantage of Stage is, is that it easily allows us to modify the maps and robots we used. Stage is a 3D-robot simulator, and therefore it enables the user to create a 3D-model of an environment. The easy integration of Stage and ROS allows for an easy transfer of data and program onto the real Turtlebots. A more complete description of ROS can be found in appendix I

4 Methodology

This section will describe the methodology used for the project. The given pseudo code describes the code used in the experiments.

4.1 Random Walk

The random walk[1] allows the robots to walk around randomly while avoiding collision. If the robot is close to another robot, it will ensure the robots enter *Chicken* mode. pseudo code to avoid obstacles is provided in Algorithm 1.

[1]The full code can be found in appendix II

Algorithm 1 Obstacle Avoidance

```
scan ← get laser scanner array
closest ← min(scan)
left_frontier ← sum(scan[0:len(scan)/2])
right_frontier ← sum(scan[len(scan)/2:end])
if closest > 80cm then
    if rand < 0.2 then
        mode ← "CCW" or "CW"
    else
        mode ← "STRAIGHT"
    end if
else
    if mode ="STRAIGHT" then
        if left_frontier < right_frontier then
            mode ← "CW"
        else
            mode ← "CCW"
        end if
    end if
end if
```

The pseudo code for the chicken mode is provided in Algorithm 2, this method gets called as soon as the robot is detected by the server, and initialises the chicken game. The twist command used is a command to publish the velocity and rotation to the robots: twist(linear speed, angular speed).

However, as cooperating can also be disadvantageous, the robots do not move when turning. In order to prevent the defectors from being stuck the whole time and therefore not being able to participate in any other game, a waiting time was added to the defectors HALT_TIME. When walking around, the waiting time is zero. As soon as the defectors get stuck in a collision and are unable to move, the waiting time increases. After a certain amount of time has passed, they are allowed to start moving again and try to free themselves form the collision. The waiting serves as a punishment for their defecting.

Algorithm 2 Chicken Game

```
if HALT_TIME < 500 then
    if mode = "STRAIGHT" then
        twist ← (0.8,0)
    end if
    if mode = "CCW" then
        twist ← (0,1)
    end if
    if mode = "CW" then
        twist ← (0,-1)
    end if
end if
if chicken_mode then
    if COOPERATION then
        if turn_direction = 0 then
            turn_direction ← choice([-1,1])
            twist ← (0, turn_direction)
        else
            if opponent_distance < 0.8 then
                twist (0,0)
            end if
        end if
    else
        turn_direction ← 0
    end if
else
    twist ← (-0.3, 20 × choice([-1,1]))
    halt_time ← HALT_TIME - 0.05
end if
Apply twist to robot
```

4.2 Global Positioning

In order to distinguish between collision with obstacles and with other robots, a blackboard coordination model[2] is used. In a blackboard model, a server collects the positions and tells the robots which ones should enter in a social interaction with others[3].

Algorithm 3 show how the server manages all the data gained from the robots, and determines which robots are close to each other. If they are within a certain distance, they get added to the neighbour list, which is then

[2]http://en.wikipedia.org/wiki/Blackboard_model
[3]The full code can be found in appendix III.

used by the robots individually to play the game. The program also continuously collects the robot's data and saves in in a simple text file, to be used by Matlab for the experiments.

Blackboard

Algorithm 3 Blackboard Coordination

```
counter ← 0
for pose in rostopic do
    R ← new Robot
    R.index ← counter
    R.x ← spose.position.x
    R.y ← pose.position.y
    R.θ ← atan(pose.orientation.z / pose.orientation.w) * 360 / π
    APPEND R to Robot_list
    counter ← counter + 1
end for
for R1 in Robot_list do
    for R2 in Robot_list except R1 do
        dist ← sqrt((R2.x − R1.x)^2 + (R2.y − R1.y)^2)
        Δθ ← atan2(R2.y − R1.y, R2.x − R1.x) * 180/π
        if dist < 1m and |Δθ − R1.θ| < 75° then
            APPEND [R1.index, R2.index, dist] TO Blackboard
        end if
    end for
end for
PUBLISH BLACKBOARD
```

4.3 Aggregating Results

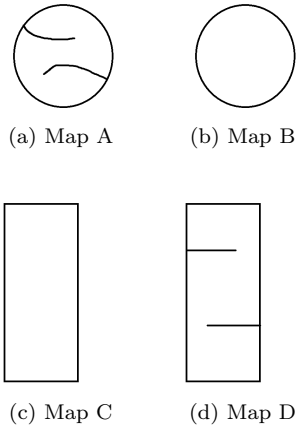
s Matlab was used to calculate the robot's fitness and plot the data into graphs. Using Matlab it is determined whether the robots moved with respect to their previous position. If they did, their score will increase. The fitness is calculated by the function described in Eq. 1.

4.4 World Map

The world maps are made using paint and GIMP. Stage simulator has the capability of building a 3D map from a 2D drawing by using a .WORLD file.

In the experiments, four maps are used as shown in Figure 1:

- Map A: This map is a circle, with two walls to keep to robots close together and test their obstacle avoidance.
- Map B: This map is a simple circle, such that the robots mix in all directions.
- Map C: This map is a simple rectangle, ensuring that the robots meet, the sharp corners however, make walking around for the robots more difficult.
- Map D: This is a map with obstacles, such that it will be more difficult for the robots to avoid collision.



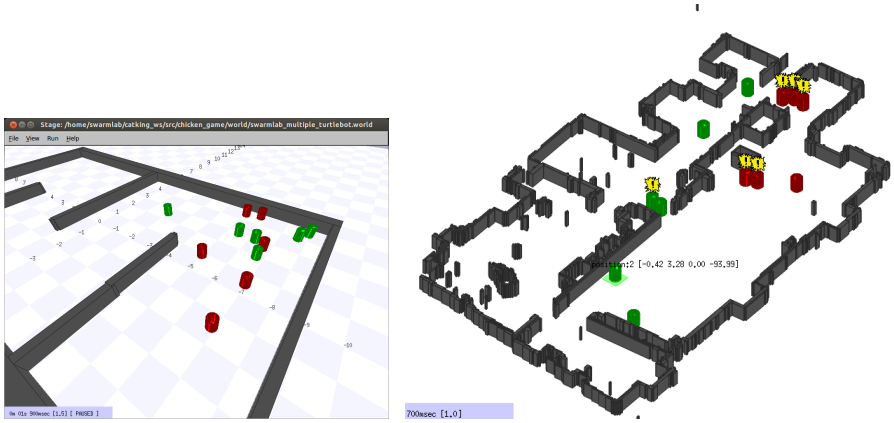
(a) Map A (b) Map B

(c) Map C (d) Map D

Figure 1: The main four maps used for experiments

5 Experiment Design

Various experiments were conducted, with various amounts of defectors and cooperators, and different maps. These experiments were run in the Stage simulator, using ROS. Using a simple ROS-publisher, the robot data was published into a text file. As the chicken game was used as the format to represent the prisoner's dilemma, being stuck in a collision can be seen as a result from defecting.



(a) (b)

Figure 2: Swarm Behaviour in a predefined map: green and red robots denote the cooperating and defecting robots respectively.

5.1 Simulation Experiments

This section will describe the experiments done in the Stage simulator.

- Experiment 1: The defectors and cooperators start on different sides of map A (see appendix IV for all the maps), functioning as small communities. After a while they mix. The goal of this experiment is to see how the groups do when they are all separated.
- Experiment 2: The defectors and cooperators start in two equally mixed groups on both sides of map A. The goal of this experiment is to see how the defectors and cooperators do in small mixed groups
- Experiment 3: The defectors and cooperators are equally mixed in map B. The goal of this experiment is to see how the cooperators do when they are mixed in a simple non-challenging environment.
- Experiment 4: There is one defector versus 11 cooperators in map B. The goal of this experiment is to see if a defector profits from being the only one to defect.
- Experiment 5: There is one cooperator versus 11 defectors in map B, the goal of this experiment is to see how the cooperator will do in a selfish environment.
- Experiment 6: There is an equal amount of defectors and cooperators in a bit more challenging map. (Map C) The goal is too see if it influence the performance of the defectors and cooperators.
- Experiment 7: There is an equal amount of defectors and cooperators in a challenging map. (Map D) The goal is too see if it influence the performance of the defectors and cooperators.
- Experiment 8: 1 defector versus 11 cooperators in map C, this is similar to experiment 4, but now the environment is more of a challenge.
- Experiment 9: 11 defectors versus 1 cooperator in map C, this is similar to experiment 5, but now the environment is more of a challenge.
- Experiment 10: 1 defector versus 11 cooperators in map D, this is similar to experiment 4, but now the environment is even more challenging.

- Experiment 11: 11 defectors versus 1 cooperators in map D,this is similar to experiment 5, but now the environment is even more challenging.

It is expected that the defectors will do worse as the difficulty of the environment increases. Also, the cooperators should do better in general, except when there are only a few defectors.

6 Results

In all experiments, the world maps are scaled to the size $10.900m \times 17.750m \times 0.500m$. The laser scanner of the robot detects the range $0m$ to $10m$, its field of view is 230.25 degrees, the number of rays are 481. The robot has the following dimensions $35cm \times 35cm \times 45cm$ (x y z), which is identical to a real Turtlebot.

The experiments introduced in previous section are all implemented in the Stage simulator and results are provided in Figures 3-13.

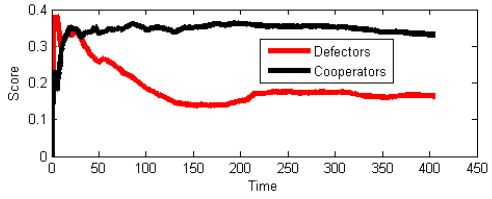


Figure 3: The results from experiment 1, in the beginning the defectors peak, but after a few seconds the cooperators have higher average score.

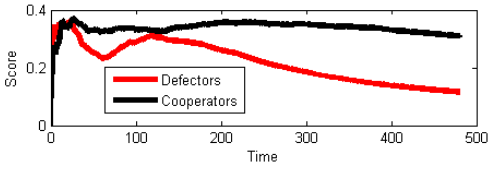


Figure 4: The results from experiment 2, In this experiment the cooperators starts with a lower score, but after a short period of time they overtake the defectors again.

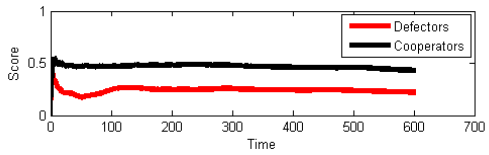


Figure 5: The results from experiment 3. The defectors and cooperators keep the same difference in score for the duration of the experiment. The cooperators are doing better than the defectors.

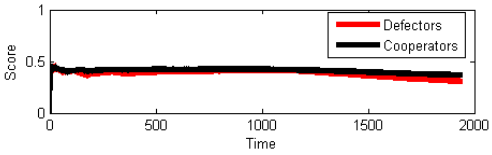


Figure 6: The results from experiment 4. The defector does slightly worse than the cooperators, but in this case the difference is minimal.

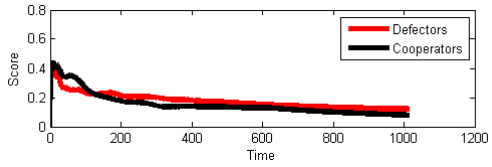


Figure 7: The results from experiment 5. The cooperator seems to do better at the beginning, but the defectors are doing overall

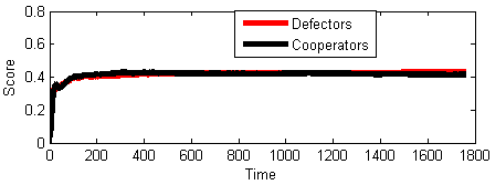


Figure 8: The results from experiment 6. The cooperators and defectors have similar scores, the cooperators are doing marginally better.

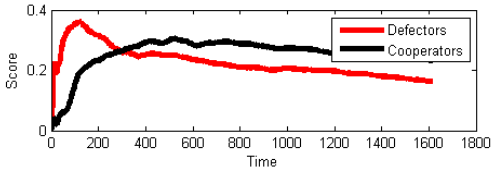


Figure 9: The results from experiment 7. The defectors do very well in the beginning, but after a while the cooperators have a better score.

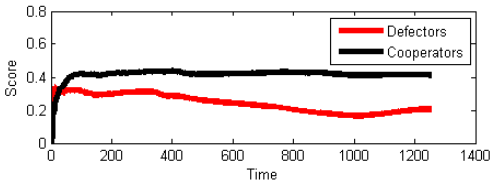


Figure 10: The results from experiment 8. The cooperators are still at an advantage, even though there is only one defector.

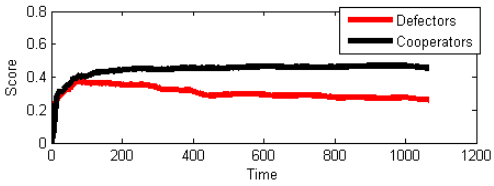


Figure 11: The results from experiment 9. The cooperator still has a higher score than the other defectors.

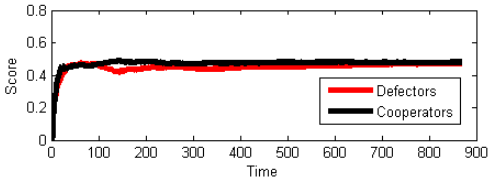


Figure 12: The results from experiment 10. The cooperators seem to do just a bit better than the one defector.

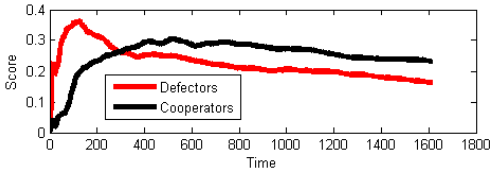


Figure 13: The results from experiment 11. The defectors seem to do better at the beginning but after a while the cooperator has a better score.

Figure 14: A heatmap showing the ratio between the cooperators and defectors score per amount of defectors and map complexity. No data was created for two scenarios: Map A and 11 defectors and Map A and 1 defector.

7 Discussion

This section refers to the research questions introduced in first section and it will discuss how this project has provided answers to each of those research questions.

RQ 1 What is an appropriate two-robot experimental setting which can represent the simple PD problem? The chicken game as described in Subsection 3.2 is a appropriate setting to represent the prisoner's dilemma.

RQ 2 How to extend the setting designed for the two-robot PD to a Swarm Robotic experiment? The robots still play the game with only two players, but because of the random walk the robots can play with anyone.

RQ 3 What is the effect of environment structure on swarm outcomes?

In general we can conclude that the defectors seem to do well in the beginning, but after a certain period of time their scores decrease. The

cooperators on the other hand, seem to need some time to get going, but afterwards their score steadily increases, allowing them to surpass the defectors. This can be explained in the following way: in the beginning, the defectors roam around freely, and the cooperators move around them, sometimes standing still to avoid them. After a while however, the defectors will get stuck in a collision with another defector, causing them to stop moving. When this has happened they become less of an obstacle for the cooperators, allowing them to roam around freely, as can be seen in experiment 1 and 2.

In a community in which the majority are defectors, the cooperators are at an advantage, as the defectors all massively collide. However, in a community in which most are cooperators, a defector will have the advantage as the cooperators will go out of their way for the defectors. In a community with a large amount of defectors, it is also possible for a cooperator to become stuck in between all the defectors. This is also in agreement with the results we see in experiment 5. However, if the cooperator is able to avoid the defectors he will have the advantage (experiment 11).

The microscopic behaviour of one defector in a community full of cooperators will result in a bigger pay-off for the defector, while it will only slightly influence the macroscopic behaviour of the whole group's pay-off, as seen in experiment 5 as well.

It seems that the defectors fail more as soon as the difficulty of the maps increases (experiment 7), the cooperators on the other hand, seem to do fine. The change in the results can be explained in a similar way as the other results, the defectors now have even more difficulty staying out of collisions, they tend to get stuck a lot. The cooperators take more time now to avoid collision, but they rarely get stuck.

RQ 4 Can robots be used to study social behaviour?

So far the robots have behaved conform our expectations. The results agree with the theory that cooperation should yield better overall fitness for the cooperators. From this it can be concluded that robots can be used to study social behaviour.

8 Future Work

There are many possibilities for future work. A simple learning algorithm could be added, such that the robots can learn from their opponent's behaviour and change their behaviour depending on the feedback they get.

Different strategies can then be used to see how they influence the robot's behaviour, if for example some strategies might be more beneficial for one robot, but not for the group as a whole. One well known strategy is tit-for-tat, meaning that the robot will recreate the behaviour form the opponent it last played with. This combined with having a small change to defect back to default behaviour or even learning will yield interesting results. This is also more applicable to human social behaviour, as humans are not likely to have a very strict set of rules as robots do.

Different maps forcing the robots to interact more or less will also have an interesting effect on the outcome, especially if the experiments can also be done with more robots.

If the experiments were done with more robots, it will be more clear how the effect of a few defectors or cooperators in a big group influence the total fitness of the group. In some experiments, one cooperator gets stuck between all the defectors, and that has a high impact on the average score as there are only a few of them.

It would also be interesting to see how the program would run on real robots, as there is in general quite a big difference between simulation and reality. Robots may have bugs or some small malfunctioning that might influence their performance.

9 Conclusion

Examining the effect of the physical environment has always been a challenge in studying

social behaviours. Therefore, in this paper the application of swarm robotics to study social behaviour was introduced. The proposed framework uses ROS and Turtlebots as they are more convenient for this kind of research allowing to use the same code on both simulators and real robots. From the experiments conducted, it can be concluded that the environment does have an influence on the performance of the robots, as was suggested earlier. It seems that robots can be used to study social behaviour, as the results are verifications of what is expected from similar types of social interactions.

10 Acknowledgements

The author would like to thank her family for their support and also to Nera Nesic for the help with Python.

References

[1] Alers, Sjriek, Tuyls, Karl, Ranjbar-Sahraei, Bijan, Cleas, Daniel, and Weiss, Gerhard (2014). Insect-inspired robot coordination: Foraging and coverage. *14th International Conference on the Synthesis and Simulation of Living Systems (ALIFE 14)*.

[2] Birk, Andreas and Wiernik, Julie (1996). Behavioral AI experiments and economics. *Workshop Empirical AI, 12th European Conference on AI, Budapest*.

[3] Birk, Andreas and Wiernik, Julie (2002). An N-player prisoner's dilemma in a robotic ecosystem. *Robotics and Autonomous Systems*, Vol. 39, No. 3, pp. 223–233.

[4] Grimaldi, Bruno (2011). Agent-based vs. robotic simulation: a repeated prisoner's dilemma experiment. M.Sc. thesis, UNIVERSIT DEGLI STUDI DI TORINO.

[5] Keller, Laurent, Wischmann, Steffen, Floreano, Dario, Mitri, Sara, et al. (2012). Using robots to understand social behavior. Technical report, Wiley-Blackwell.

Appendix I - ROS documentation

A Introduction

Robots Operating System, also known as ROS is a framework to write robot software.

A.1 How ROS works

Topics & Nodes

A ROS node is an executable that is connected to the ROS-network and communicates with other nodes. Nodes can publish messages on topics. ROS nodes are designed to work on a small topic. One node may control the robot's laser scanner, while another may control the wheel.

The nodes communicate with each other by publishing messages onto topics. When running Turtlesim, a simulator, in ROS, and moving the turtle (the robot) via the keyboard, the keyboard node will publish the incoming keystrokes onto a topic, which will that be read by the Turtlesim node to move the turtle.

Using the command line, the user can get an overview of all the active nodes and topics. The user can also use the command line to subscribe to a topic and get all the messages in it. The messages in ROS can all have different formats, this must be noted when trying to publish messages.

Publishers and Subscribers

The information published in all the ROS topics can be viewed and used by using publishers and subscribers. As stated above, when running the turtlesim node and using the keyboard to move the turtle, the keyboard node publishes the keystrokes to the turtle's command_velocity topic. The turtlesim simulator is subscribed to that topic, meaning that it will receive the messages published into that topic.

A.2 Rospy

The user can write code to subscribe and publish onto various topics. This can be done either via C++ or Python. This paper will mainly focus on Python.

In order to be able to communicate with ROS via Python, the Python code needs to import the Rospy module.

The topic and messages can be imported into the program and then the information contained in those messages can be used in the code.

Appendix II - stage_random_walk.py

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
import random as RD
from std_msgs.msg import String

global robot_number, mode, chicken_mode

mode = "STRAIGHT"
chicken_mode = False
```

```
MAX_SPEED = 0.8
MAX_ROTATION_SPEED = 1
import random as RD

COOPERATION = True
MIN_DISTANCE = 0.8 # obstacle avoidance distance, for simulator
 #.5 works just fine while for real robots we should stick to
 #something like 1.0
halt_time = 0;
opponent_distance = 1000; # initialize with a large number

def GetLaser(msg):
        GetLaser2(msg,robot_number)

def GetLaser2(msg, robot_number):
        global mode, closest, MIN_DISTANCE


        HALF_RANGE = len(msg.ranges)/2  # half of the laser resolution

        scan = msg.ranges # a list of all scanned ranges

        closest = min(scan[HALF_RANGE / 3 : 5 * HALF_RANGE / 4])
        # distance to the closest obstacle
        closest_index = min( (v, i) for i, v in enumerate(scan) )[1]
        # index of closest obstacle
        furthest_index = max( (v, i) for i, v in enumerate(scan) )[1]
        # index of the furthest obstacle

        if closest > MIN_DISTANCE: # if there is no obstacle near by
                if RD.random() < .2:
                        mode = RD.choice(["TURN_CCW", "TURN_CW"])
                else:
                        mode = "STRAIGHT"
        else:
                if mode == "STRAIGHT":
                        if sum(scan[:HALF_RANGE]) < sum(scan[HALF_RANGE:]):

        # if there is an obstacle, check which side there are less obstacles
                                mode = "TURN_CCW"
                        else:
                                mode = "TURN_CW"
        talker(mode,robot_number)


def CollisionDetect(msg):
```

```
        global chicken_mode, opponent_distance, halt_time

        blackboard = eval(msg.data)
        col_list = blackboard['neighbors_list']
        temp_flag = False
        for pair in col_list:
                if int(robot_number) is  pair[0]:
                        temp_flag = True
                        opponent_distance = pair[2]


        if temp_flag:
                chicken_mode = True
                halt_time = halt_time + 1

        else:
                chicken_mode = False
                halt_time = 0

def listener():
        rospy.init_node('listener',anonymous=True)
        rospy.Subscriber('robot_'+ robot_number + "/base_scan", LaserScan, GetLaser)
        rospy.Subscriber("/chatter", String, CollisionDetect)
        rospy.spin()

def set_twist(x,z):
        twist = Twist()
        twist.linear.x = x  # our forward speed
        twist.linear.y = 0
        twist.linear.z = 0 # we can't use these!

        twist.angular.x = 0
        twist.angular.y = 0
        twist.angular.z = z # rotate CCW

        return twist

def talker(mode, robot_number):

        global chicken_turn_direction, halt_time

        pub = rospy.Publisher('robot_' + robot_number + '/cmd_vel',Twist)

        if halt_time < 500:
                if mode == "STRAIGHT":
                        twist = set_twist(MAX_SPEED,0)
```

```
                if mode == "TURN_CCW":
                        twist = set_twist(0,MAX_ROTATION_SPEED)
                if mode == "TURN_CW":
                        twist = set_twist(0,-1*MAX_ROTATION_SPEED)
                if chicken_mode:
                        if COOPERATION:
                                if not chicken_turn_direction:
                                        chicken_turn_direction = RD.choice([-1,1])
                                twist = set_twist(0,chicken_turn_direction *
                                MAX_ROTATION_SPEED)
                        else:
                                #print opponent_distance, MIN_DISTANCE
                                if opponent_distance < MIN_DISTANCE :
                                        twist = set_twist(0,0)
                else:
                        chicken_turn_direction = 0;

        else:

                twist = set_twist(-.3, RD.choice([-1,1]) * 20 * MAX_ROTATION_SPEED)
                halt_time = halt_time - 0.05
        pub.publish(twist)


if __name__ == '__main__':
        global robot_number
        import sys
        try:
#               robot1 = sys.argv[1]
#               robot2 = sys.argv[2]
#               print sys.argv[1][-1]
                robot_number = sys.argv[1][:-1]
                robot_type = sys.argv[1][-1]
        except:

                print 'ERROR: no robot_name provided, try: \n$$ python
        stdr_random_walk.py robot[XX] '
                exit()
        if robot_type == 'C':
                print 'robot', robot_number, 'started to wander around as
        a COOPERATOR'
                COOPERATION = True
        else:
                print 'robot', robot_number, 'started to wander around as
        a DEFECTOR'
                COOPERATION = False
```

```
        listener()
        rospy.spin()
```

Appendix III - talker.py

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String
from nav_msgs.msg import Odometry
from rosgraph_msgs.msg import Clock
import math as MT
import time
import os.path

save_path = "/home/bija/Dropbox/#PAPERS/Simulations/Turtlebot_Scripts/matlab/data"
name_of_file = time.strftime("%c")
name= os.path.join(save_path, name_of_file+".txt" )

f = open(name, 'w')

CHICKEN_DIST = 1

pose_dict = {}

blackboard = {}
rostime = 0




def get_pose(msg):
        global rostime
        pose = {}
        pose['sin'] = msg.pose.pose.orientation.z
        pose['cos'] = msg.pose.pose.orientation.w
        pose['theta'] = MT.atan(msg.pose.pose.orientation.z/
        msg.pose.pose.orientation.w)/MT.pi*360
        pose['x'] = msg.pose.pose.position.x
        pose['y'] = msg.pose.pose.position.y
        robot_id = msg.header.frame_id[7:-5]
        pose_dict[robot_id] = pose
        f.write(str(rospy.get_time()) + " " + str(robot_id) + " "
        +str (pose['x']) + " " +str (pose['y']) + " "
        +str (pose['theta'])  + "\n")
```

```
def talker():
        pub = rospy.Publisher('chatter', String)
        rospy.init_node('talker', anonymous=True)

        for i in xrange(100):
                rospy.Subscriber('robot_' + str(i) +
                        "/base_pose_ground_truth",
                        Odometry, get_pose)
        r = rospy.Rate(10) # 10hz
        while not rospy.is_shutdown():
                neighbors_list = []
                for key1 in pose_dict.keys():
                        for key2 in pose_dict.keys():
                                if key2 is not key1:
                                        x1 = pose_dict[key1]['x']
                                        y1 = pose_dict[key1]['y']
                                        theta1 = pose_dict[key1]['theta']
                                        x2 = pose_dict[key2]['x']
                                        y2 = pose_dict[key2]['y']
                                        if x2 == x1:
                                                x2 = x2 + .01;
                                        relative_theta = MT.atan2((y2 - y1),(x2-x1))
                                        /MT.pi*180
                                        if MT.fabs(x1 - x2) < CHICKEN_DIST
and
                                        MT.fabs(y1-y2)
                                        < CHICKEN_DIST and MT.fabs(relative_theta
                                        - theta1) < 75:
#                                               print relative_theta, theta1
                                                neighbors_distance = (MT.fabs(x1-x2)
                                                MT.fabs(y1-y2))
                                                neighbors_list.append(
                                                [int(key1),int(key2),
                                                neighbors_distance ])

#                                       if int(key1) == 0 and int(key2) == 1:
#                                               print theta1, relative_theta

                #str_variable = "hello world %s"%rospy.get_time()
                # rospy.loginfo(str_variable)
                # pub.publish(str(pose_dict)
                blackboard['neighbors_list'] = neighbors_list
                pub.publish(str(blackboard))
```

```
                r.sleep()

if __name__ == '__main__':
    try:
        talker()
        f.close()
    except rospy.ROSInterruptException: pass
```