UNIVERSITY OF ZAGREB

**FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING**

# MULTI-ROBOT SYSTEMS

## Controlling a Swarm of Sphero Robots in a Simulator Stage using Reynolds Rules

*Drmić Jakov, Jurković Marin, Kozulić Josip Ante,*
*Pušnik Filip, Škoro Filip, Vuksan Ivan*

Zagreb, December 2022.

## Content

# 1. Introduction

The goal of this project was to implement algorithms based on Reynolds rules to control Sphero robots in simulation. The implemented algorithms were tested in a simple 2D simulator Stage in Robot Operation System (ROS). The results of the simulation will be shown and explained later in the report.

As already mentioned, the main goal was to implement three algorithms based on Reynolds rules; ***separation*** - boids try to move away from nearby flock mates to avoid collisions, ***alignment*** - boids try to match their velocity and heading with other boids and ***cohesion*** - boids try to move closer to the other boids to form a flock. These three rules alone are sufficient to make the robots group and move randomly together.

Reynolds flocking rules are shown in figure [Figure 1]:

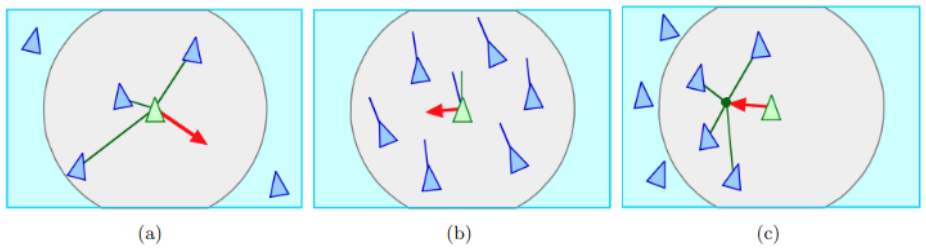


*Figure 1 Separation, Alignment and Cohesion*

Each boid has direct access to the whole scene's geometric description, but flocking requires that it reacts only to flockmates within a certain small neighborhood around itself. The neighborhood is characterized by a distance (measured from the center of the boid) and an angle, measured from the boid's direction of flight, as shown in figure [Figure 2]:

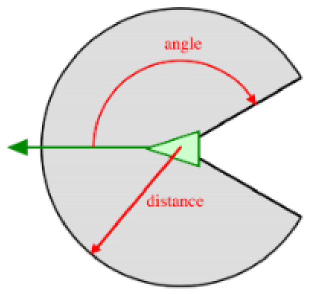


*Figure 2 Field of View*

The second part of this project was to implement two additional Reynolds rules; ***navigation to a certain point*** - where the whole swarm needs to reach given point in the environment and ***obstacle avoidance*** - where swarm or any member of it should avoid obstacles in the environment.

Sphero robots are simulated in Stage without mass which means robots can achieve the desired velocity instantly.

Velocities are set up by sending a message on the certain ROS topic and at the same time, the simulator is continuously streaming positions and velocities of the robots to the ROS topics.

Task implementation has been simulated on different maps and also some parameters have been changed to observe different affect on the final result. Example of map in simulation is shown in figure (Figure 3).

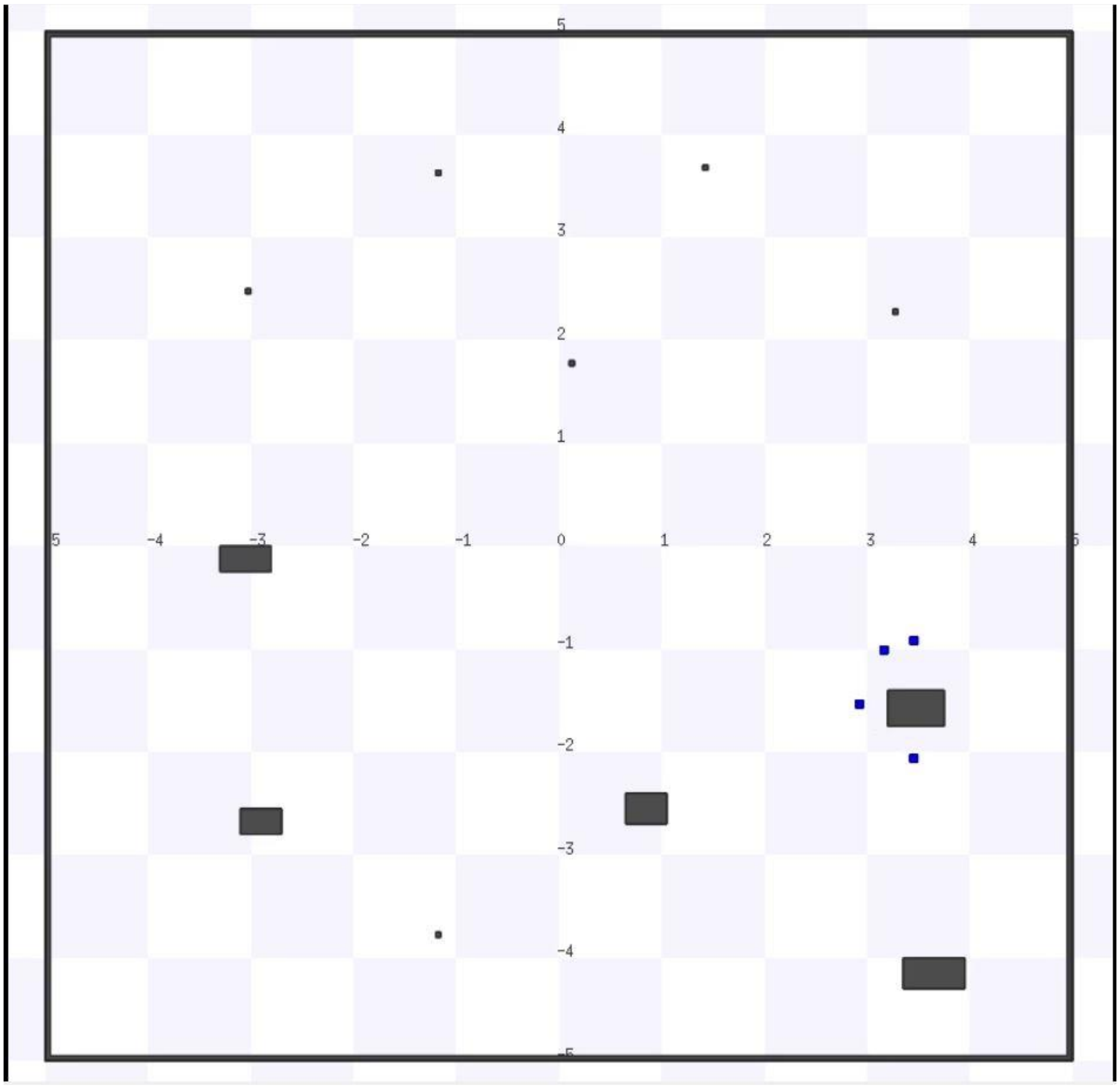


*Figure 3 Map in simulation*

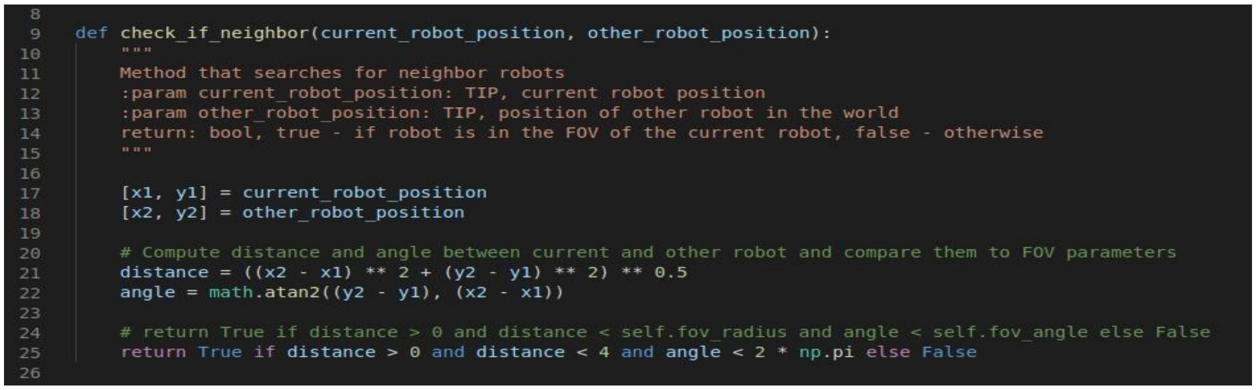
## 2. Assignment

Assignment has two main parts. In the first part of assignment our task was to implement and simulate three basic Reynolds rules: separation, alignment, cohesion in Stage simulator. The goal of the second part was to implement two additional Reynolds rules: navigation to a given point and obstacle avoidance.

## 2.1. Implementation

This subchapter briefly explains implementation of each Reynolds rules algorithms.

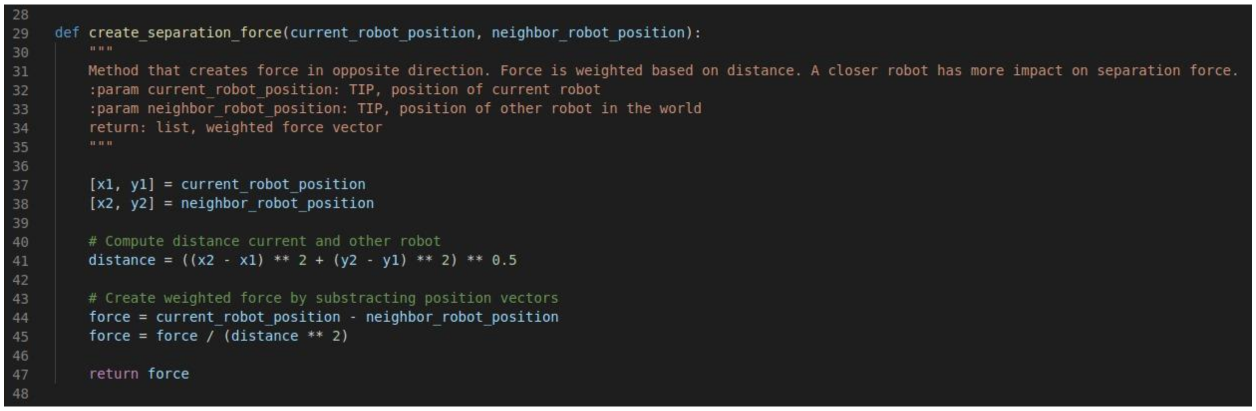
The first step of each algorithm was to find each boids neighbors. That was solved in the function *check_if_neighbor,* which is shown in figure (Figure 4). Input arguments for this function are current boid position and other boid position and by using those arguments it calculates distance and angle between two boids. Based on those results, it decides whether those boids are neighbors or not.



```python
def check_if_neighbor(current_robot_position, other_robot_position):
    """
    Method that searches for neighbor robots
    :param current_robot_position: TIP, current robot position
    :param other_robot_position: TIP, position of other robot in the world
    return: bool, true - if robot is in the FOV of the current robot, false - otherwise
    """

    [x1, y1] = current_robot_position
    [x2, y2] = other_robot_position

    # Compute distance and angle between current and other robot and compare them to FOV parameters
    distance = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
    angle = math.atan2((y2 - y1), (x2 - x1))

    # return True if distance > 0 and distance < self.fov_radius and angle < self.fov_angle else False
    return True if distance > 0 and distance < 4 and angle < 2 * np.pi else False
```

*Figure 4 Check if neighbor function*

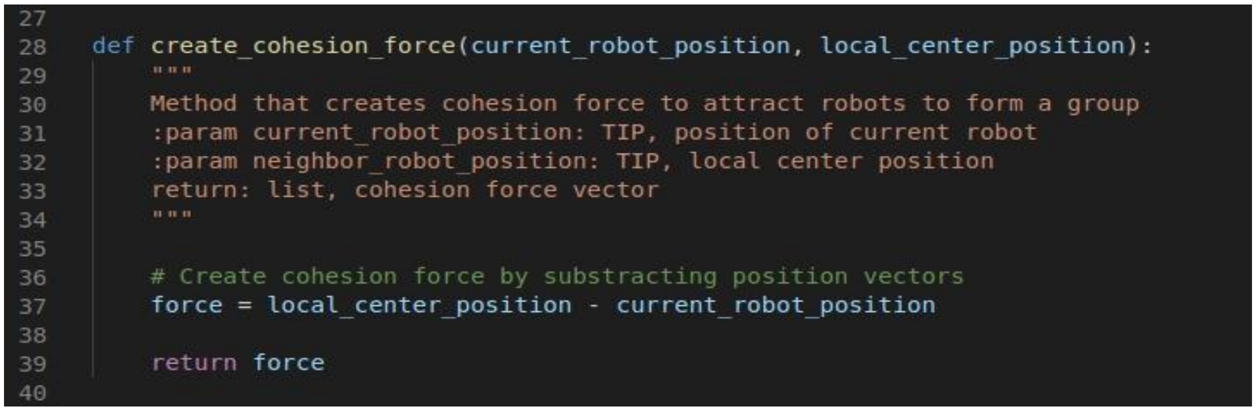
If current boid has neighbors than it is possible to proceed with algorithms. Main part of separation algorithm is implemented in *create_separation_force* function which gets current boid position and neighbor boid position as inputs. Function calculates separation force by subtracting boid's positions which is then weighted based on squared distance. Function is shown in figure (Figure 5).



```python
def create_separation_force(current_robot_position, neighbor_robot_position):
    """
    Method that creates force in opposite direction. Force is weighted based on distance. A closer robot has more impact on separation force.
    :param current_robot_position: TIP, position of current robot
    :param neighbor_robot_position: TIP, position of other robot in the world
    return: list, weighted force vector
    """

    [x1, y1] = current_robot_position
    [x2, y2] = neighbor_robot_position

    # Compute distance current and other robot
    distance = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5

    # Create weighted force by substracting position vectors
    force = current_robot_position - neighbor_robot_position
    force = force / (distance ** 2)

    return force
```

*Figure 5 Separation force function*

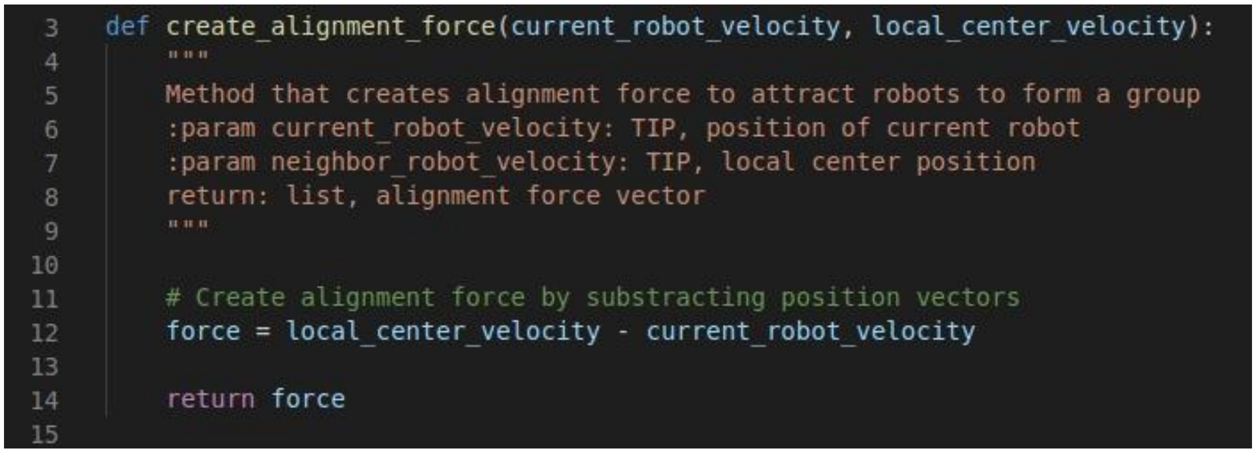
Cohesion algorithm is implemented in function *create_cohesion_force*. It gets current boid position and position of local center as inputs. Local center is center of mass calculated by averaging sum of boid's neighbors positions. Cohesion force is calculated by subtracting current robot position from local center. Function is shown in figure (Figure 6).



```python

def create_cohesion_force(current_robot_position, local_center_position):
    """
    Method that creates cohesion force to attract robots to form a group
    :param current_robot_position: TIP, position of current robot
    :param neighbor_robot_position: TIP, local center position
    return: list, cohesion force vector
    """

    # Create cohesion force by substracting position vectors
    force = local_center_position - current_robot_position

    return force

```

Figure 6 Cohesion force  function

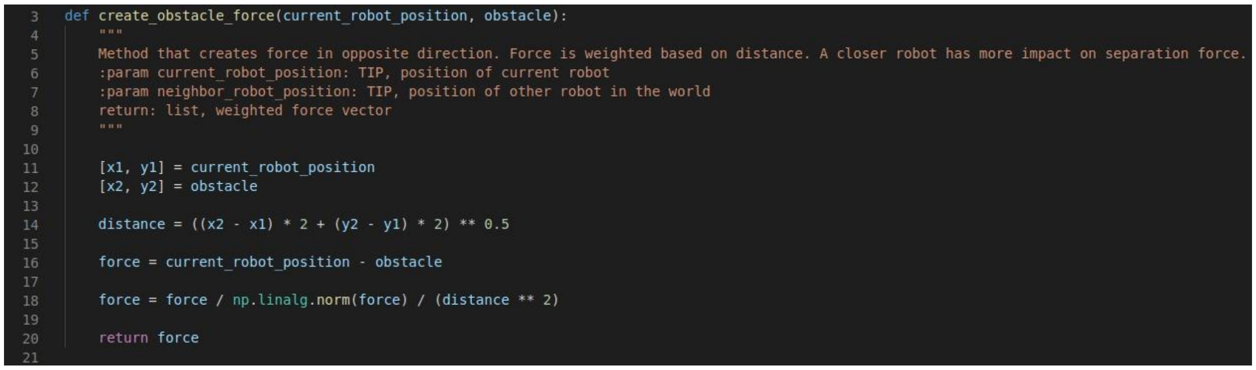
Alignment algorithm is implemented in function *create_alignment_force*. This function gets current boid velocity and local center velocity as inputs where local center velocity is averaging sum of boid's neighbors velocities. Alignment force is calculated by subtracting current robot velocity from local center velocity. Function is shown in figure (Figure 7).



```python
def create_alignment_force(current_robot_velocity, local_center_velocity):
    """
    Method that creates alignment force to attract robots to form a group
    :param current_robot_velocity: TIP, position of current robot
    :param neighbor_robot_velocity: TIP, local center position
    return: list, alignment force vector
    """

    # Create alignment force by substracting position vectors
    force = local_center_velocity - current_robot_velocity

    return force

```

Figure 7 Alignment force function

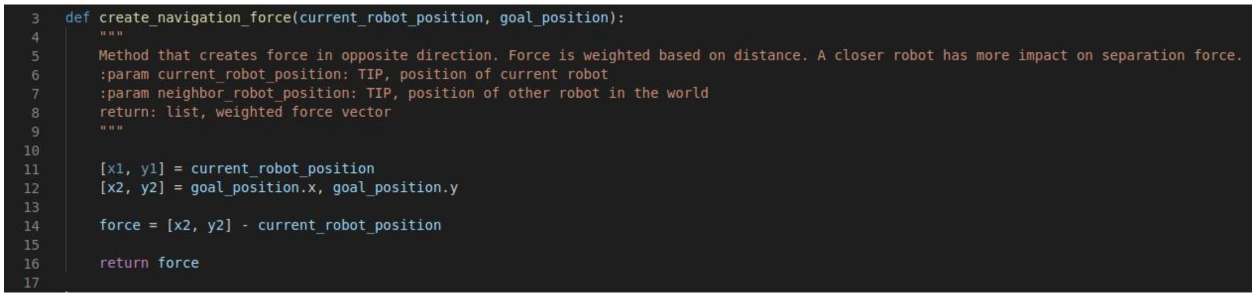
Obstacle avoidance is achieved by function create_obstacle_force. This function gets current boid position and obstacle position as input arguments and then it calculates force by subtracting obstacle position from current boid position after which result is divided by the norm of the force vector and then again that result is divided by squared distance. Function is shown in figure (Figure 8).



```python
def create_obstacle_force(current_robot_position, obstacle):
    """
    Method that creates force in opposite direction. Force is weighted based on distance. A closer robot has more impact on separation force.
    :param current_robot_position: TIP, position of current robot
    :param neighbor_robot_position: TIP, position of other robot in the world
    return: list, weighted force vector
    """

    [x1, y1] = current_robot_position
    [x2, y2] = obstacle

    distance = ((x2 - x1) * 2 + (y2 - y1) * 2) ** 0.5

    force = current_robot_position - obstacle

    force = force / np.linalg.norm(force) / (distance ** 2)

    return force
```

*Figure 8 Obstacle avoidance function*

Navigation to a certain point is achieved by create_navigation_force function. This function gets input arguments current boid position and goal position where goal position is desired boid's position. It calculates force by subtracting current boid position from goal position. Function is shown in figure (Figure 9).



```python
def create_navigation_force(current_robot_position, goal_position):
    """
    Method that creates force in opposite direction. Force is weighted based on distance. A closer robot has more impact on separation force.
    :param current_robot_position: TIP, position of current robot
    :param neighbor_robot_position: TIP, position of other robot in the world
    return: list, weighted force vector
    """

    [x1, y1] = current_robot_position
    [x2, y2] = goal_position.x, goal_position.y

    force = [x2, y2] - current_robot_position

    return force
```

*Figure 9 Navigation force function*

## 2.2. Simulation

This subchapter shows our results of simulation. As mentioned earlier, simulations of algorithms were carried out in 2D simulator Stage in ROS. This subchapter also shows how different parameters influence the system, but that part is explained in detail later in the subchapter.

First Video shows simple map with no obstacles and boids separating from each other, creating a group and then moving together in the same direction.

Second Video shows boids moving on a simple map with very few and small obstacles. Every implemented algorithm is included and everything seems to be working correctly.

Third Video shows boids moving on the same map, but this time alignment wasn't included. It is obvious that boids don't move correctly and even two of them collided.
Fourth Video shows a more complicated situation; map is a simple maze with walls that represent obstacles. Boids had to move from one corner to another and they moved successfully.
Fifth Video shows even more complicated maze. Once again boids had to move from one corner to another and once again they moved successfully.
Sixth Video and last one, shows boids moving on a simple map, but this time they have a leader which is followed by other boids.
These videos shows how implemented algorithms affect boids on a different maps. Everything went as expected; boids moved correctly where everything implemented was included and they didn't move correctly where some parts of assignment where omitted.

## 3. Conclusion

As was stated before, the goal of the project was to implement algorithms based on Reynolds rules aiming to simulate the movement of groups of animals. Simulations were carried out by using simple Sphero robots and the implemented Reynolds rules were tested in the 2D simulator Stage. By simulating Sphero robots as simple omni-directional points, it was achieved that they have no inertia, i.e. they can achieve the comanded velocity instantly.

After simulating all of the three main implemented rules, it is obvious that separation and cohesion algorithms are complementary; separation algorithm tends to disperse the group of boids while cohesion tends to make them move in a group. Alignment algorithm is different from separation and more similar to cohesion because it also keeps boids in a group, but it also makes them move in the same direction. All algorithms proved to be efficient.

In addition to the three main Reynolds rules, two more rules have been implemented; obstacle avoidance and navigation to a certain point. By navigating boids through the maze, obstacle avoidance algorithm proved to be working correctly as all boids reached the goal. So it is possible to conclude that both algorithms are efficient as shown in a simulation.

After the performed simulations and presented results, it can be said that the initial assumptions are correct, that is, that the implemented algorithms work as expected.