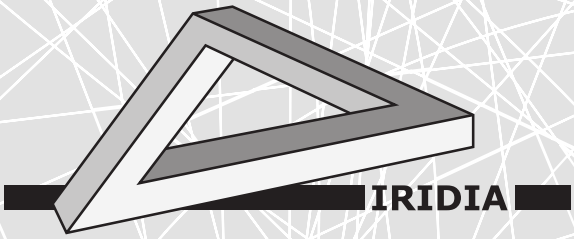


Université Libre de Bruxelles

Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

Mercator: hardware and software architecture for experiments in swarm SLAM

M. Kegeleirs, R. Todesco, D. Garzón Ramos, G. Legarda Herranz, and M. Birattari

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2022-012

November 2022
Last revision: November 2022

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:

IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2022-012

Revision history:

TR/IRIDIA/2022-012.001 November 2022
TR/IRIDIA/2022-012.002 November 2022

Mercator: hardware and software architecture for experiments in swarm SLAM

Miquel Kegeleirs, Raffaele Todesco,
David Garzón Ramos, Guillermo Legarda Herranz, and Mauro Birattari

IRIDIA, Université libre de Bruxelles, Belgium.

November 2022

1 Introduction

The Sphero RVR is an education-tailed device packed with a large sensor set. Its modularity makes it an interesting candidate for robotics experiments, especially in swarm robotics. These native new capabilities, such as odometry – the ability for the robot to locate itself into space – and the extended autonomy might be a qualitative addition to the existing robot pool.

This document presents Mercator, an upgraded version of the Sphero RVR, as well as an ecosystem for experimentation in swarm robotics with Mercator, including simulation in ARGoS3 and a cross-platform control software interface that enables transparency between simulated and real-life behaviors of the robot.

2 Hardware architecture

The factory version of the Sphero RVR is equipped with the following sensors and actuators:

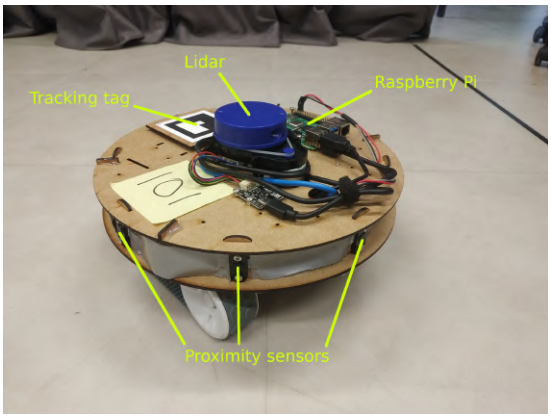
- an RGB ground color sensor that detects the color of the floor;
- a velocity sensor;
- a 3-axis accelerometer;
- a 3-axis gyroscope, measuring angular velocities;
- an IMU sensor, that measures the robot pitch, roll and yaw;
- an ambient light sensor;
- a locator odometry sensor, that allows the robot to locate itself;
- 5 RGB LEDs;
- 2 independent treads to move the robot.

Mercator is paired with a Raspberry Pi 4[1], and is equipped with additional sensors:

- 8 proximity sensors (Terabee Teraranger Multiflex[2]);
- a rotating lidar (YDLIDAR X4 [3]).

[1] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/
[2] https://www.terabee.com/shop/lidar-tof-range-finders/teraranger-multiflex/
[3] https://www.ydlidar.com/products/view/5.html

(a) Base Sphero RVR



(b) Mercator

Figure 1: The base Sphero RVR and Mercator.

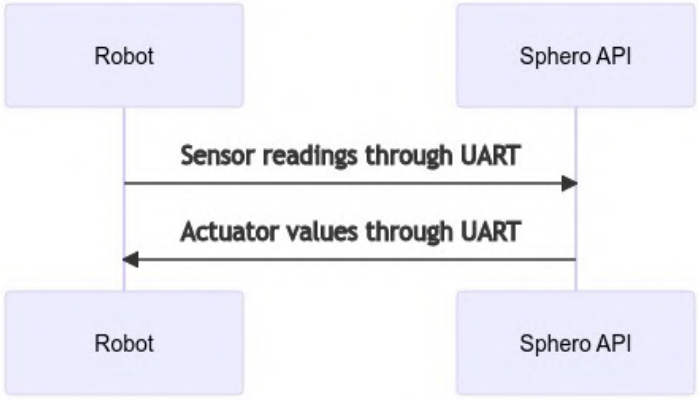


Figure 2: Mercator software architecture

Communication between the robot and the Raspberry Pi is ensured through the UART protocol.

As shown on Figure 1b, the additional components (lidar, proximity sensors, and Raspberry Pi) are mounted on a cardboard "hat" on top of the robot. Consequently, the ground and odometry sensors are still fully working, while the robot is able to detect obstacles with the proximity sensors. The lidar is the only element at its height level in the experimental environment, in order to enable neighbor detection between the robots. At the moment, the robot is however unable to detect ambient light. This will be fixed in future improvements as the opaque cardboard will be replaced by translucent Plexiglas.

3 Software architecture

Sphero developed a Python API[4] to enable interaction between the Rasperry Pi and Mercator through the UART protocol. Any Linux distribution such as Ubuntu[5] or Raspberry Pi OS[6](previously Raspbian) supports this API. This represents the first level of abstraction presented in Figure 2.

This base architecture allows to design control software for the robot, but only within this Python environment which, as such, is not compatible with the ARGoS3 simulator. Hence, we use ROS to enable communication between the ARGoS3 C++ control software and the Python RVR driver. In particular, the control software will subscribe to sensor topics and publish actuator values, and the driver will do

[4] https://github.com/sphero-inc/sphero-sdk-raspberrypi-python
[5] https://ubuntu.com/download/server
[6] https://www.raspberrypi.com/software/operating-systems/

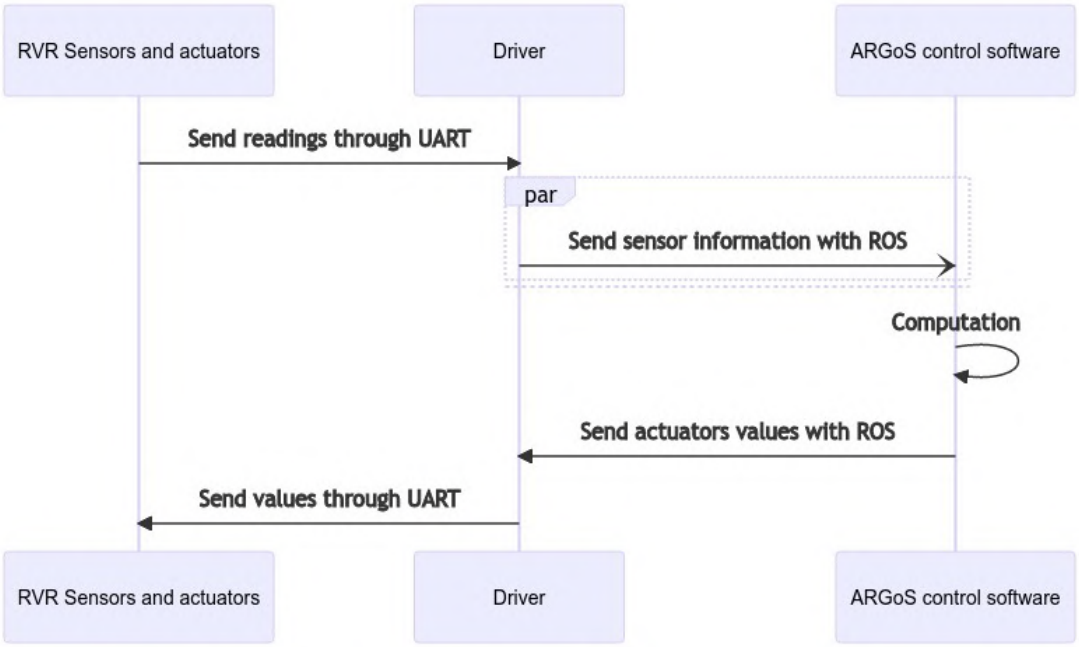


Figure 3: Mercator - ARGoS3 architecture with ROS

the opposite.

The following topics are used to share information between the driver and the control software:

- $/rvr/ground_color$: the color detected by the ground sensor;
- $/rvr/ambient_light$: the ambient light perceived;
- $/rvr/imu$: IMU information, that groups:
  - The orientation (pitch, roll and yaw) provided by the IMU sensor;
  - Angular velocities from the gyroscope;
  - Linear acceleration from the accelerometer.
- $/rvr/odom$: odometry information, that groups:
  - The *pose*, including the position estimated by the locator and the orientation provided by the so-called quaternion sensor;
  - The *twist*, i.e., the angular velocities of the gyroscope and the linear velocities from the velocity sensor.
- $/rvr/wheels_speed$: the speed of the robot's treads;
- $/rvr/rgb_leds$: the color of the LEDs

This architecture is presented in Figure 3.

The flexibility and modularity of this architecture is highlighted when adding the additional hardware modules: the lidar and the proximity sensors. Indeed, these sensors have their own ROS nodes (software components), and they can connect to the same ROS master system as well. In order to integrate the lidar and the proximity sensors with the rest of the architecture, they publish their information on dedicated ROS topics (respectively, $/scan$ and $/ranges$), which can be read by the ARGoS3 control software. Hence, no additional work on the architecture is required to incorporate new sensors or actuators to the robot.

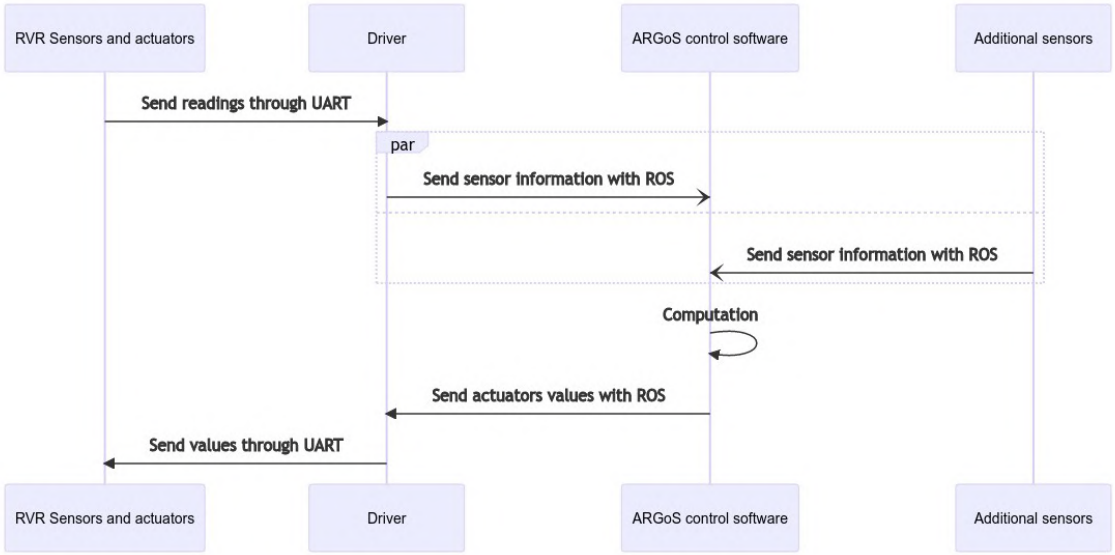


Figure 4: Mercator complete architecture

The final architecture is schematized in Figure 4.

This modular architecture also helps to attenuate a ground sensor calibration issue. Depending on the robot, the color sensed by the robot beneath is highly biased, either towards darker or lighter tones. As no official calibration procedure is provided by Sphero, a color labeler ROS node has been integrated. This additional node maps the raw reading to the RGB value of the closest label, defined by :

$$\text{label} = \text{argmin}_i \; d_i \tag{1}$$

$$d_i = (R_{sensor} - R_i)^2 + (G_{sensor} - G_i)^2 + (B_{sensor} - B_i)^2 \tag{2}$$

The result of this labeling is output on its own topic ($/rvr/fixed_color$) which is provided as an input to controllers.

4 Simulation environment

4.1 Robot model

A 3D model of the Sphero RVR has been realized, according to robot measures and pictures. This 3D model can be seen in Figure 5. Regarding physics, a 2D physics model is implemented, using a cubed colliding box around the robot shape, similarly to the e-puck implementation[7]. The treads are undergoing a classical differential-drive model, with 2 distinct speeds v_l and v_r.

4.2 Simulating sensors

Due to the large sensor set of Mercator, simulating its capabilities accurately was a substantial part of this work.

The implementation of each sensor in simulation is described in the following sections, including the noise evaluation and generation.

4.2.1 Ground color sensor

The ARGoS3 simulator already includes grayscale ground sensor in some robot implementations. However, Mercator is able to detect the color underneath it and provide it as an RGB value. In ARGoS3, this is simulated by checking the color of the virtual floor, at the position of the projection of the virtual

[7] `https://github.com/demiurge-project/argos3-epuck`

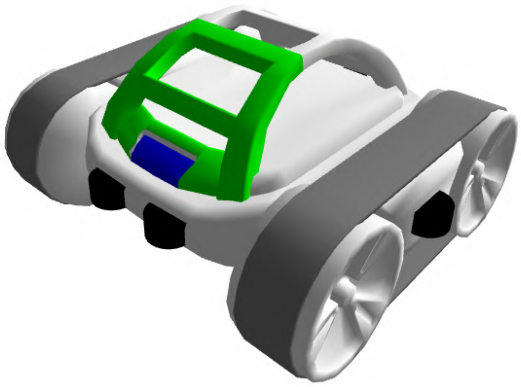


Figure 5: Sphero RVR 3D model. Black cylinders are visual representations of the RGB LEDs. The green part of the upper side highlights the front of the robot.

color sensor on it. This virtual floor can be set as any image by the user.

The noise of the sensor has been evaluated for 4 different benchmark color measurements: red, green, blue and yellow, over 300 samples each. The measured noise histograms, as well as their kernel density estimation, are presented in Figure 6. Accordingly, a noise following a Gaussian distribution $N(0, 5)$ for each color channel is recommended in the experiment configuration. The μ and σ parameters of the Gaussian distribution can be tweaked by the user to better simulate its own experimental environment.

4.2.2 Velocity sensor

The velocity sensor provides the robot speed along the X and Y axis "absolute" axis, in m/s. The X and Y axis here are defined as the axis according to Figure 7, at the boot time of the robot. However, ARGoS3 rigidbodies only provide the position and orientation of the robot in the virtual space, the speed must hence be computed from wheel speeds v_l and v_r:

1. Compute the velocity v of the robot by averaging v_l and v_r.
$$v = \frac{v_l + v_r}{2} \tag{3}$$
2. Find the relative orientation Q_{rel} from the original absolute orientation Q_i and the current orientation Q_c, where all orientations are represented by quaternions.
$$Q_{rel} = Q_i^{-1} * Q_c \tag{4}$$
3. Compute the Euler angle α, representing the rotation around the Z axis, from Q_{rel}.
4. Return the speed vector as
$$\vec{v} = (v \cos \alpha, v \sin \alpha) \tag{5}$$

4.2.3 Accelerometer

The 3-axis accelerometer provides the robot acceleration around its 3 axis (see Figure 7), in g. The axis are defined as in the velocity sensor, the same method is hence applied to find the relative orientation Q_{rel}. To be able to simulate acceleration, the evolution of speed over time is computed from v_l and v_r:

1. Compute the velocity v of the robot by averaging v_l and v_r, as earlier.
$$v = \frac{v_l + v_r}{2} \tag{6}$$

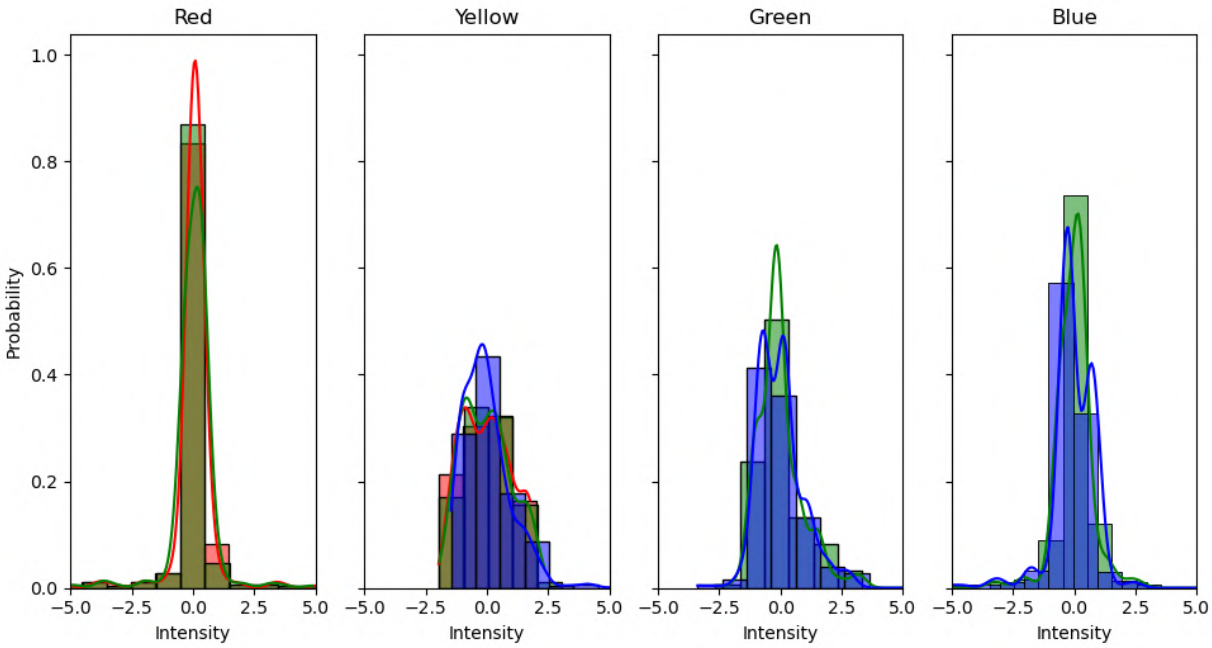


Figure 6: Noise of the color sensor for several ground colors. Each histogram color represents the corresponding color channel.

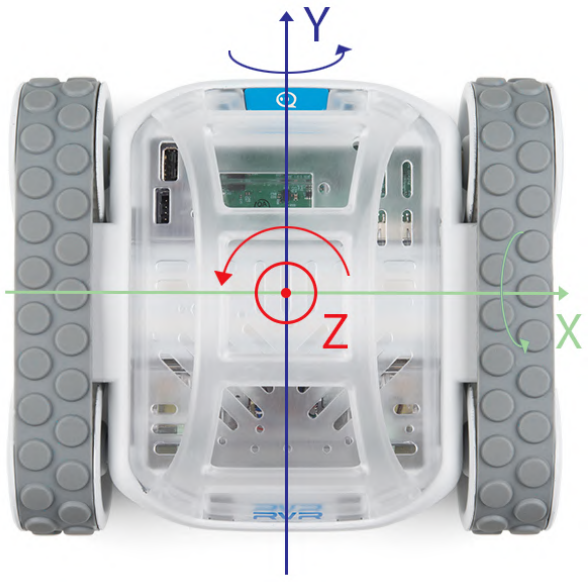


Figure 7: Mercator axis system

2. Compute the relative orientation Q_{rel} and find the Euler angle α from it.

$$Q_{rel} = Q_i^{-1} * Q_c \tag{7}$$

3. Compute the current velocity vector $\vec{v}_t$ from α.

$$\vec{v}_c = (v \cos \alpha, v \sin \alpha) \tag{8}$$

4. Retrieve the time spent between 2 speed measurements, which is the clock tick of ARGoS physics simulator Δt.

5. Compute the acceleration vector $\vec{a}$, with the previous speed vector $\vec{v}_{t-1}$, and convert it to g ($G = 9.81\ m/s^2$).

$$\vec{a} = \frac{\vec{v}_t - \vec{v}_{t-1}}{G\Delta t} \tag{9}$$

6. Update the value $\vec{v}_{t-1}$ to $\vec{v}_t$ for the next accelerometer reading.

4.2.4 Gyroscope

The gyroscope provides the angular velocity of the robot around each axis. Similarly to the accelerometer, we only have access to the current orientation of the robot, and not its angular speed, which requires to compute the difference between 2 timesteps:

1. Retrieve the differential orientation Q_t, the rotation between the previous timestep orientation Q_{t-1} and the current one Q_c.

$$Q_t = Q_c * Q_{t-1}^{-1} \tag{10}$$

2. Convert Q_t to Euler angles α_X, α_Y and α_Z, which represent the relative angular rotation and constitute the angular vector $\vec{\alpha}$.

3. Retrieve again the time spent between 2 measurements Δt.

4. Compute the angular velocity vector $\vec{v}$:

$$\vec{v} = \frac{\vec{\alpha}}{\Delta t} \tag{11}$$

5. Update Q_{t-1} to Q_t.

4.2.5 IMU

The Inertial Measure Unit (IMU) abstracts information from the magnetometer, accelerometer and gyroscope of the robot to provide orientation as pitch, roll, and yaw angles. As the IMU measurement is based on an external reference (with the magnetometer) that all robots share, they also share the same referential system for the IMU. For that reason, the simulated IMU provides the orientation that comes from the absolute referential of the simulator, considering it has a virtual North pole that robots can align to.

4.2.6 Light sensor

The single ambient light sensor is placed at the front of the robot and can perceive the ambient light in lux. To simulate it, it is required to sum the contribution of each light source in the scene and take occlusion into account:

1. Go through every light source in the scene

2. Check that light is not occluded by raycasting

3. Distribute the reading to the sensor according to its distance to it and its intensity

4. Clamp the value to the range the sensor can detect

The sum of all light sources contributions then constitute the sensor reading.

4.2.7 Locator

The locator can provide odometry information as (X, Y) values inside an axis system that is defined as the robot is aligned to North pole, similarly to the IMU. Although the axis systems of each robot are aligned, the origin of the axis is the original position of each robot:

1. At the beginning of the experiment, save the position of the robot as the origin of the locator axis (X_o, Y_o)
2. Provide (X, Y) readings as the current coordinates of the robot in the absolute simulator axis system (X_c, Y_c), translated to the locator origin.

$$(X, Y) = (X_c, Y_c) - (X_o, Y_o) \tag{12}$$

4.2.8 Quaternion

The quaternion sensor provides the relative orientation of the robot with respect to its booting position. The approach is similar to the locator's:

1. At the beginning of the experiment, save the orientation of the robot Q_i
2. Find the relative orientation Q_{rel} from the original absolute orientation Q_i and the current orientation Q_c

$$Q_{rel} = Q_i^{-1} * Q_c \tag{13}$$

4.2.9 Proximity sensors

The proximity sensors allow to detect obstacles between 0.05 and 2 meters. To detect obstacles in simulation, a raycasting approach has been applied:

1. For each sensor, check the first occluding object in the direction of the sensor
2. If the distance between the sensor and the obstacle is in the range $[0.05, 2]$, the sensor reading is that distance. Otherwise, it is $-\infty$ or $+\infty$ if the object is too close or too far, respectively.

The noise of the proximity sensors is normally-distributed with a low deviation, as shown in Figure 8.

4.2.10 Lidar

The lidar, due to its very similar functionality, has been implemented in the same way as the proximity sensors, with different sensor placements, many more readings (719 against 8), and a broader range : [0.10, 12] meters.

The lidar noise is distributed as in Figure 9. Accordingly, a uniform noise has been integrated, to which the user can tweak the range to better fit its observations and environment.

4.3 Simulating actuators

The current actuator set comprises :

1. 4 wheels grouped into 2 treads (connecting 2 wheels each);
2. 5 RGB LEDs (2 in the front of the robot, one on each side, one in the back).

In simulation, the wheels are considered as a differential drive to which 2 different speeds can be submitted (left and right), which matches the actual robot implementation. The LEDs are represented with black cylinders (see Figure 5) and can display any color from its RGB encoding.

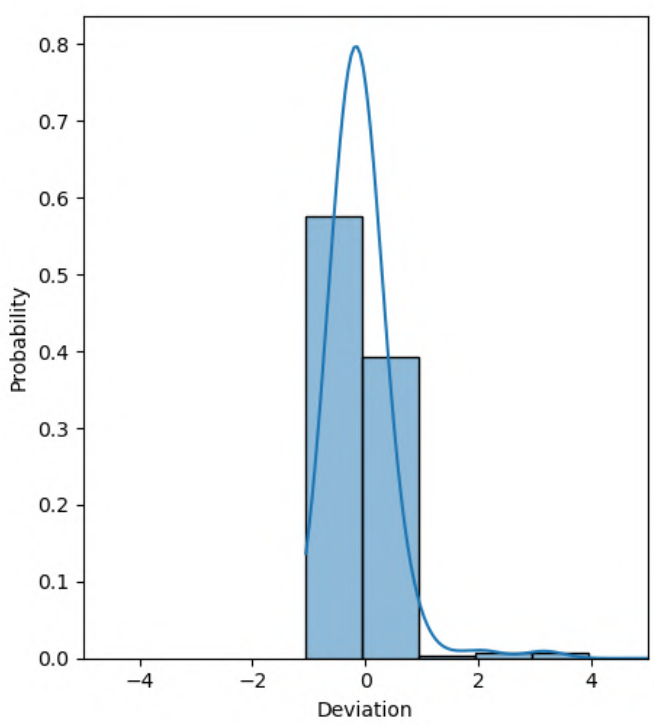


Figure 8: Proximity sensors noise distribution

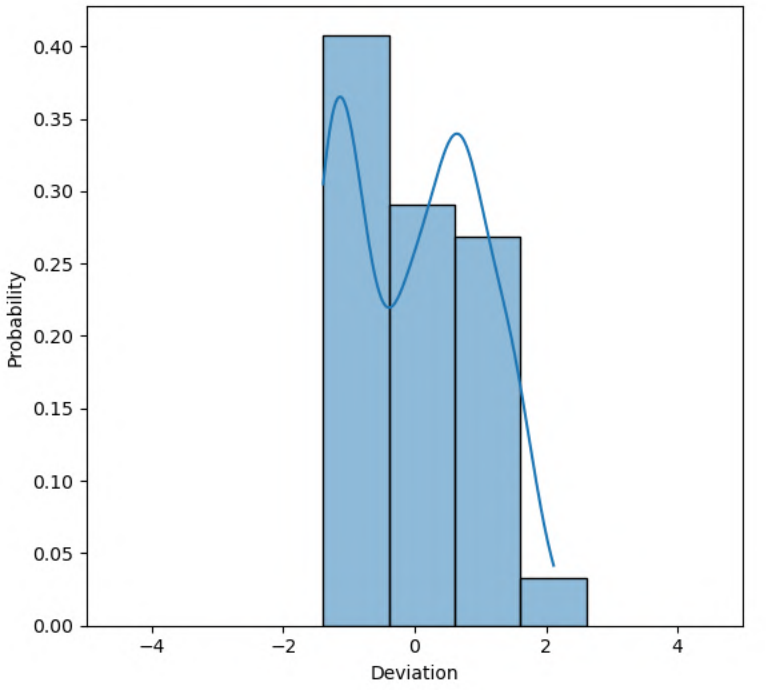


Figure 9: Lidar noise distribution

Sensor/actuators	Variables	Range of values
Proximity	$prox$	$[0,1]^8$
	$\angle q$	$[0,2\pi)^8$
Light	$light$	$[0,16000]$
Ground color	R,G,B	$[0,255]^3$
Neighbour localization	d	$[0,10]^m, m \in \mathbb{N}$
	$\angle q$	$[0,2\pi)^m, m \in \mathbb{N}$
Wheels	v	$[0,155]^2$

Table 1: The reference model variables for the RVR

4.4 The reference model

In AutoMoDe, every robot needs to be described as a *reference model*, a set of input/output variables that are used by the different modules of the state machines produced by AutoMoDe. They provide another layer of abstraction between the control software and the sensors and actuators. The reference model variables for the Mercator are presented in Table 1.

The proximity sensor provides 8 readings included in $[0,1]$, 1 meaning that an obstacle is less than 0.05 meters away from the sensor, and 0 signalling no obstacles in range. The maximum range of the proximity sensor can be tuned by the user to face different use cases. The proximity sensor also indicates the angle of the sensor with respect to the head of the robot.

The light sensor provides the illuminance it perceives, in lux. However, as the latest robot prototype does not have access to ambient light, this sensor is currently not in use.

The ground color sensor, as expected, provides the color output by the color labeler as an RGB value triplet.

The neighbor locator is an abstraction that is required because of the difference between the robot implementation and the simulation. To detect its neighbors, the real robot uses the lidar readings and clusters them into groups it will consider as a single robot. In that way, each robot can detect its neighbors in a 0 to 10 meters range by detecting the lidar of other robots which is the only object at the same height. However, due to the physics simulation model, simulating inter-lidar detection is not possible. The simulation uses an omnidirectional camera that is able to detect virtual, invisible LEDs placed on the top of other robots. Practically, these two ways of detecting neighbors aim at the same objective, which are grouped under the neighbour localization sensor of the reference model.

To extract the position of neighbors from lidar readings, they are clustered in the following way :

1. The algorithm goes through every of the 719 lidar readings.
2. If a reading lies outside of the [0.10, 0.75] range, it is discarded because it can not belong to another robot. The upper bound of 75 cm is thought to avoid considering walls of the room in which the arena lies as neighbors.
3. If 2 consecutive readings have a similar distance and angle (with arbitrary thresholds set at 2 cm and 10 degrees), they are clustered together and considered as belonging to the same robot.

This algorithm eventually outputs the number of neighbors, their distance and their heading with respect to the current robot.

The wheel variables are trivially the speed of each tread.

4.5 AutoMoDe-*Watermelon*

AutoMoDe-*Chocolate* control software takes the form of a probabilistic finite state machine (PFSM), which is composed of 2 types of modules :

1. *Behaviors* : These are the states in the PFSM and represent a core behavior the robot can accomplish : exploration, stop, attraction/repulsion to neighbors or light, etc.

2. *Transitions* : The transitions are the conditions to transition from one state to the other. These can be fixed probability, color of the floor, neighbor count, etc.

In particular, AutoMoDe-*Chocolate* contains the same behavior modules as the original AutoMoDe-*Vanilla*, which are the following Francesca et al. (2014):

1. *Exploration* : the robot moves straight. If any of the proximity sensors positioned in front senses an obstacle, that is, if $prox_i \geq 0.1$ for any $i \in 1, 2, 7, 8$, the robot turns on itself for a random number of control cycles chosen in 0,1,...,τ,where τ is an integer parameter in 1,2,...,100. The robot turns away from the direction faced by the proximity sensor that returned the highest value.

2. *Stop* : the robot stays still.

3. *Phototaxis* : the robot moves towards the light source, if perceived; otherwise, it moves straight. Obstacle avoidance is embedded: the robot follows the vector $\mathbf{w} = \mathbf{w}' - k\mathbf{w}_o$, where k is a hard-coded parameter whose value has been *a priori* fixed to 5 and $\mathbf{w}'$ and $\mathbf{w}_o$ are vectors defined as:

$$\begin{aligned} \mathbf{w}' &= \begin{cases} \mathbf{w}_l = \sum_{i=1}^{8} \left(\text{light}_i, \angle q_i\right), & \text{if light is perceived.} \\ (1, \angle 0), & \text{otherwise;} \end{cases} \\ \mathbf{w}_o &= \sum_{i=1}^{8} \left(prox_i, \angle q_i\right), \end{aligned} \tag{14}$$

4. *Anti-phototaxis* : the robot moves away from the light source, if perceived; other-wise, it moves straight. Obstacle avoidance is embedded: the robot follows the vector $\mathbf{w} = \mathbf{w}' - k\mathbf{w}_o$, where

$$\mathbf{w}' = \begin{cases} -\mathbf{w}_l, & \text{if light is perceived} \\ (1, \angle 0), & \text{otherwise:} \end{cases} \tag{15}$$

   and k, $\mathbf{w}$', and $\mathbf{w}_o$ are defined in phototaxis.

5. *Attraction* : the robot goes in the direction of the robots in neighborhood, if any; otherwise, it moves straight. Obstacle avoidance is embedded: the robot follows the vector $\mathbf{w} = \mathbf{w}' - k\mathbf{w}_o$, where

$$\mathbf{w}' = \begin{cases} \mathbf{w}_n = \sum_{m=1}^{n} \left(\frac{\alpha}{d_i}, \angle q_i\right), & \text{if robots are perceived} \\ (1, \angle 0), & \text{otherwise;} \end{cases} \tag{16}$$

   and α is a real-valued parameter in [1,5], and where $\mathbf{w}_o$ and k are defined in phototaxis.

6. *Repulsion* : the robot moves away from the other robots in its neighborhood, if any; otherwise, it moves straight. Obstacle avoidance is embedded: the robot follows the vector $\mathbf{w} = \mathbf{w}' - k\mathbf{w}_o$, where

$$\mathbf{w}' = \begin{cases} -\mathbf{w}_n, & \text{if robots are perceived} \\ (1, \angle 0), & \text{otherwise;} \end{cases} \tag{17}$$

   and $\mathbf{w}_n$ is defined in attraction, while $\mathbf{w}_o$ and k are defined in phototaxis.

However, the (anti-)phototaxis behaviors rely on the presence, for the e-puck, of multiple light sensors which allow to find the direction of the light with respect to the robot. As Mercator owns a single light sensor, these behaviors can not be implemented without regularly sampling the light intensity around the robot to find its direction. Moreover, as the current prototype obstructs the light sensor, these modules will not be used with Mercator. Exploration, stop, attraction and repulsion are compatible and working with this robot.

Nonetheless, these modules also had to be tweaked, as the original modules were designed for the e-puck. This leads to the introduction of AutoMoDe-*Watermelon*, the first AutoMoDe flavor designed for Mercator. The behavior modules of AutoMoDe-*Watermelon* are the following :

1. *Exploration* : the robot moves straight. If any of the proximity sensors positioned in front senses an obstacle, that is, if $prox_i \geq 0.1$ for any $i \in 1, 2, 7, 8$, or if there is a neighbor in front, the robot turns on itself for a random number of control cycles chosen in 0,1,...,τ,where τ is an integer parameter in 1,2,...,100. The robot turns away from the direction faced by the proximity sensor that returned the highest value.

2. *Stop* : the robot stays still.

3. *Attraction* : the robot goes in the direction of the robots in neighborhood, if any; otherwise, it moves straight. Obstacle avoidance is embedded: the robot follows the vector $\mathbf{w} = k_w\mathbf{w}' - k_o\mathbf{w}_o$, where
$$\mathbf{w}' = \begin{cases} \mathbf{w}_n = \sum_{m=1}^{n} \left(\frac{\alpha}{d_i}, \angle q_i\right), & \text{if robots are perceived} \\ (1, \angle 0), & \text{otherwise;} \end{cases} \tag{18}$$
and α is a real-valued parameter in [1,5], and where $\mathbf{w}_o$ is defined in phototaxis; k_w and k_o are hard-coded hyperparameters that have been *a priori* set respectively to 1.2 and 5.

4. *Repulsion* : the robot moves away from the other robots in its neighborhood, if any; otherwise, it moves straight. Obstacle avoidance is embedded: the robot follows the vector $\mathbf{w} = \mathbf{w}' - k\mathbf{w}_o$, where
$$\mathbf{w}' = \begin{cases} -\mathbf{w}_n, & \text{if robots are perceived} \\ (1, \angle 0), & \text{otherwise;} \end{cases} \tag{19}$$
and $\mathbf{w}_n$ is defined in attraction, while $\mathbf{w}_o$ and k are defined in phototaxis.

Regarding transition conditions, AutoMoDe-*Watermelon* contains :

1. *Black floor* : if the ground is black (ground reading in grayscale is lower than 0.05), the transition is enabled with probability β, where β is a parameter.

2. *Gray floor* : same as black-floor but the prerequisite is that the ground reading in grayscale is around 0.5.

3. *White floor* : same as black-floor but the prerequisite is that the ground reading in grayscale is above 0.78.

4. *Neighbor count* : the transition is enabled with probability
$$z(n) = \frac{1}{1 + e^{\eta(\xi - n)}} \tag{20}$$
where n is the number of robots in the neighborhood, $\eta \in [0, 20]$ is a real-valued parameter and $\xi \in 0, 1, ..., 10$ is an integer parameter. The transition is enabled with probability 0.5 if $n = \xi$. The parameter η regulates the steepness of the function $z(n)$ at $n = \xi$.

5. *Inverted neighbor count* : the transition is enabled with probability $1 - z(n)$, where $z(n)$ is defined in neighbor count.

6. *Fixed probability* : the transition is enabled with probability β, where β is a parameter.

All of these transitions are compatible and implemented with Mercator. Moreover, to take advantage of the ground color sensor, a 7^{th} transition has been created, inspired by AutoMoDe-*Tuttifrutti*Garzón Ramos and Birattari (2020):

7. *Color detection* : if the robot perceives a given color c among a fixed set (red, green, blue and yellow), the transition is enabled with probability β. β and c are parameters. To be resistant to noise, the hue of the color perceived by the sensor is compared to each of the hues of the expected labels. If it is close enough (under an arbitrary threshold of 15°), the closest label is considered as the perceived color.

An example of an AutoMoDe-*Watermelon* PFSM can be found in Figure 10.

In order to obtain a control software that highlights a desired collective behavior of the swarm, the FSM choice problem is turned into an optimization problem that will be tackled by *iterated F-race*Birattari et al. (2010), as in AutoMoDe-*Chocolate*Francesca et al. (2015). This algorithm will then maximize the performance of the swarm by evaluating different FSMs in the search space within a certain budget, i.e., the number of ARGoS3 simulations allowed. The performance of a given control software is defined by mission-specific indicators. Each FSM is restricted to a maximum of 4 states, and 4 outgoing transitions (which can not be self-transitions). Once the budget is depleted, iterated F-race outputs the best state machine it could find.

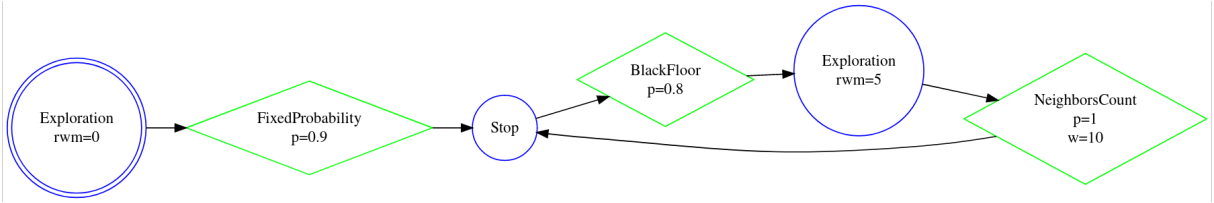


Figure 10: An example of a PFSM control software that could be designed with AutoMoDe-*Watermelon*

5 Installation

Every package is not required for simulation use. Refer to the following table for environment requirements.

Package	Required for simulation	Required for real robot
`argos3`	✓	✓
`argos3-rvr`	✓	✓
`rvr_ros`	✗	✓
RVR DAO	✓ (master branch)	✓ (ros branch)
Loop functions	✓	✓
AutoMoDe-*Watermelon*	✓ (master branch)	✓ (ros branch)

These instructions are meant to be exhaustive and executed from scratch on a brand new simulation/real robot setup. If you start from the Raspberry Pi image, you already have everything installed but perhaps not up-to-date. **Do not do anything else than updating the code and recompiling in that case**.

5.1 The `ARGoS3` simulator

This installs the ARGoS3 software **in its beta48 version**. You can refer to the full installation instructions here for the required dependencies.

```
git clone https://github.com/ilpincy/argos3 ~/argos3
cd ~/argos3
git checkout 3.0.0-beta48
mkdir build && cd build
cmake ../src
make
sudo make install
```

5.2 The `argos3-rvr` plugin

This plugin adds the RVR entity and its sensors and actuators to ARGoS.

```
git clone https://github.com/rafftod/argos3-rvr ~/argos3-rvr
cd ~/argos3-rvr
mkdir build && cd build
cmake ../src
make
sudo make install
```

5.3 The `rvr_ros` package

This package contains the driver and labelling nodes, as well as a draft controller for the real robot. You can refer to the full installation instructions here for the required dependencies.

```
git clone https://github.com/rafftod/rvr_ros rvr_ros ~/rvr_ros
cd ~/rvr_ros
git submodule init
```

```
git submodule update
rosdep install --from-paths src --ignore-src -r -y
catkin_make
```

To run the scripts you should run :

```
source ./devel/setup.bash
```

Recommended: add it to `.bashrc` to avoid doing it at each login :

```
echo "source ${PWD}/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Create a symbolic link for the Sphero SDK in the required folder :

```
ln -s ~/sphero-sdk-raspberrypi-python/sphero_sdk/ ~/rvr_ros/src/rvr_ros/src/
```

Rename the lidar port :

```
roscd ydlidar_ros_driver/startup
sudo chmod 777 ./*
sudo sh initenv.sh
```

5.4 RVR Reference model - DAO

This contains all the abstraction of the reference model.

Important : this repository is split into 2 branches :

- `master` for the simulation. Does not require ROS but does not work on the real robot. Mainly existing to be run on the cluster or other environments where ROS is not supported.
- `ros` for the real robot (or simulation if your simulation environment is compatible with ROS). Requires ROS and works with the real robot.

Installation from `master`

```
git clone https://github.com/rafftod/demiurge-rvr-dao ~/demiurge-rvr-dao
cd ~/demiurge-rvr-dao
git checkout master
mkdir build && cd build
cmake ..
make
sudo make install
```

Installation from `ros`

```
git clone https://github.com/rafftod/demiurge-rvr-dao ~/demiurge-rvr-dao
cd ~/demiurge-rvr-dao
git checkout ros
catkin_make install
```

5.5 Loop functions

This package contains the RVR loop functions.

Important : this repository is split into 2 branches :

- `master` for running the experiments normally. You should install this on the real robot and on your simulation environment.
- `epuck` to replace RVRs with e-pucks to be compatible with the tracking system. You should install this only on the computer tracking the robots.

```
git clone https://github.com/rafftod/experiments-loop-functions ~/rvr-loop-functions
cd ~/rvr-loop-functions
mkdir build && cd build
cmake ..
make
sudo make install
```

5.6 AutoMoDe-Watermelon

This contains AutoMoDe for the RVR.

Important : this repository is split into 2 branches :

- `master` for the simulation. Does not require ROS but does not work on the real robot. Mainly existing to be run on the cluster or other environments where ROS is not supported.
- `ros` for the real robot (or simulation if your simulation environment is compatible with ROS). Requires ROS and works with the real robot.

Installation from `master`

```
git clone https://github.com/rafftod/ARGoS3-AutoMoDe ~/ARGoS3-AutoMoDe
cd ~/ARGoS3-AutoMoDe
git checkout master
mkdir build && cd build
cmake ..
make
```

Installation from `ros`

```
git clone https://github.com/rafftod/ARGoS3-AutoMoDe ~/ARGoS3-AutoMoDe
cd ~/ARGoS3-AutoMoDe
git checkout ros
catkin_make
```

6 Usage

6.1 Simulation

You can simply run an experiment from an ARGoS file :

```
argos3 -c your_experiment.argos
```

Some examples figure in the `experiments` folder of the `ARGoS3-AutoMoDe` repository. Most importantly, you should adapt :

- The loop function path and label in the `loop_functions` tag
- The AutoMoDe controller path in the `automode_controller` tag :
  - `/home/USERNAME/ARGoS3-AutoMoDe/build/src/libautomode.so` for the `master` branch
  - `/home/USERNAME/ARGoS3-AutoMoDe/devel/lib/libautomode.so` for the `ros` branch

The rest is an usual ARGoS file.

6.2 Real robot

Start the lidar :

```
roslaunch ydlidar_ros_driver X4.launch
```

Start the proximity sensors :

```
rosrun teraranger_array teraranger_multiflex _portname:=/dev/ttyACM0
```

Check that ranges are actually output by the Terabees :

```
rostopic echo ranges
```

Start the color labeler :

```
rosrun rvr_ros color_labeling.py
```

Start the driver :

```
rosrun rvr_ros rvr_async_driver.py
```

Start the ARGoS controller :

```
argos3 -c your_experiment.argos
```

A real robot example figures in the `experiments` folder of the `ARGoS3-AutoMoDe` repository (`ros` branch, `real_robot.argos` file). Most importantly, you should adapt :

- The loop function path and label in the `loop_functions` tag
- The AutoMoDe controller path in the `automode_controller` tag : `/home/ubuntu/ARGoS3-AutoMoDe/devel/lib/libautomode.so` should be the correct path.

Important : to stop the robot, run :

```
sh ~/rvr_ros/kill_nodes.sh
```

This kills the `argos3` and driver processes. You have to start back the driver and ARGoS to use the robot.

Acknowledgments

This project has received funding from the Fondation Jaumotte-Demoulin and from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (DEMIURGE Project, grant agreement No 681872) and from Belgium's Wallonia-Brussels Federation through the ARC Advanced Project GbO (Guaranteed by Optimization). M. Birattari and M. Kegeleirs acknowledge support from the Belgian Fonds de la Recherche Scientifique (FNRS). D. Garzón Ramos acknowledges support from the Colombian Ministry of Science, Technology and Innovation — Minciencias.

References

Birattari, M., Yuan, Z., Balaprakash, P., and Stützle, T. (2010). *F-Race and Iterated F-Race: An Overview*, pages 311–336. Springer Berlin Heidelberg, Berlin, Heidelberg.

Francesca, G., Brambilla, M., Brutschy, A., Garattoni, L., Miletitch, R., Podevijn, G., Reina, A., Soleymani, T., Salvaro, M., Pinciroli, C., Mascia, F., Trianni, V., and Birattari, M. (2015). AutoMoDe-Chocolate: automatic design of control software for robot swarms. *Swarm Intelligence*, 9(2/3):125–152.

Francesca, G., Brambilla, M., Brutschy, A., Trianni, V., and Birattari, M. (2014). Automode: A novel approach to the automatic design of control software for robot swarms. *Swarm Intelligence*, 8(2):89–112.

Garzón Ramos, D. and Birattari, M. (2020). Automatic design of collective behaviors for robots that can display and perceive colors. *Applied Sciences*, 10(13):4654.